

Genomic **L**ocus **O**peration - **D**ata**B**ase ®

version 1.0

Installation and User Guide

*By Stephen Fisher and Junhyong Kim
Biology Department, University of Pennsylvania.*

Table of Contents:

Table of Contents:.....	2
Overview:.....	3
Installing and Running GLO-DB:.....	4
Query Language:.....	5
Command Line Interface:	6
Graphical User Interface:	7
Menu Options:	8
Usage Examples:.....	9
Working with the GUI: Compute SNPs located in conserved elements	9
Going between the GUI and CLI: Compute SNPs that are located in conserved elements or transcription factor binding sites.....	16
Working with objects in the CLI: Compute SNPs located in splice sites.....	17
Working with Sequences: Extract nucleotide sequence for each transcription factor binding site containing SNPs.....	20
Class Methods:.....	26
Track:	26
Features:	28
Sequence:	29
Query Language Examples:.....	31
Scripting Language:	34
Batch Mode:.....	35
System Requirements:	36
Known Issues:.....	36
Contact Information:.....	37

Overview:

GLO-DB is designed to perform position-based queries of genomic sequence annotations (features). It contains a query language that affords many different types of position searches via command line and graphical user interfaces, and incorporates various visualization tools. In this application, features are combined into sets, called “tracks,” where a single track can contain features from any number of genomics sequences. For example, a track might contain all exons in a genome, the introns on a particular chromosome segment, etc. These feature sets can be loaded from different types of text files but are best represented by GFF (General Feature Format) files. These files are structured so that each line encodes a single feature and thus one file can contain all the features in a track.

http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml

Since features are just start and stop positions on a sequence, each feature can be viewed as a unique object located on the sequence or as a mask over the specified region of the sequence. GLO-DB’s built-in operators will seamlessly manipulate features in either representation. For example, a user might be interested in the set of all exons on a chromosome that overlap with a specific set of genes on that same chromosome. In this case one track would contain the set of exons (“exon_track”), another the set of genes (“gene_track”). To find all overlapping features, the user would perform an “AND” operation on these two sets of features, returning a track containing the set of overlapping features (“exon_track AND gene_track”). If the user only wanted the exons in the output set, the genes could then be subtracted out (“((exon_track AND gene_track) sMINUS gene_track)”¹). Alternatively, a user could “subtract” the positions of the exons on a chromosome from the gene positions, to get a track containing a set of new features that represent the introns in the genes. Using tracks containing the exons (“exon_track”) and genes (“gene_track”), the user would then negate the two (“gene_track – exon_track”), returning a set of new features encoding the positions within the genes not encoded by the exons. In the first example, the “*set based*” operators acted on the features as immutable position pairs allowing for the sets to be altered but not the features themselves. In the second example, the “*binary*” operator acted on the features as positions on the sequence, allowing for the features to be spliced and merged into new features.

¹ The actual syntax would be: (T:exon_track AND T:gene_track) sMINUS T:gene_track

Installing and Running GLO-DB:

In order to run GLO-DB, it is necessary to first install java version 1.5 or later. If not already installed, java can download java from the Sun java website.

<http://www.java.com/en/download/manual.jsp>

Linux

Installation:

1. Unzip the package in the desired directory.
 >> unzip gloDB.zip
2. If needed adjust the attributes for the application executable.
 >> chmod a+x gloDB.sh

Running:

1. Change directories into the “gloDB” directory and run "./gloDB.sh".

Microsoft Windows 2000 or later

Installation:

1. Unzip the package in the desired directory (WinZip or some other pkzip based application will suffice).

Running:

1. From within the “gloDB” folder double-click the "gloDB.bat" application.

Mac OS X

Installation:

1. Unzip the package in the desired directory.
 >> unzip gloDB.zip
2. If needed, adjust the attributes for the application executable.
 >> chmod a+x gloDB.sh

Running:

1. Change directories into the “gloDB” directory and run "./gloDB.sh".

Query Language:

The complete list of operators is included below. In performing queries, it is possible to mix and match set and binary operators.

Set based operators:

T1 OR T2	union (symmetrical)
T1 AND T2	intersection (symmetrical) <ul style="list-style-type: none">• two features overlap if any portion of the features overlap
T1 sAND T2	intersection (symmetrical) <ul style="list-style-type: none">• sfeatures only overlap if their boundaries match exactly
T1 MINUS T2	relative compliment (asymmetrical) <ul style="list-style-type: none">• a feature in T1 will be removed from the output set if any portion of the feature in T1 overlaps with a feature in T2
T1 sMINUS T2	relative compliment (asymmetrical) <ul style="list-style-type: none">• a feature in T1 will be removed from the output set only if the features boundaries exactly match those of a feature in T2
T1 POS {x, y} T2	order (asymmetrical) <ul style="list-style-type: none">• features in T1 and T2 will be included based on their relative positions to each other and the values of x and y

Binary based operators:

T1 T2	union (symmetrical)
T1 && T2	intersection (symmetrical)
T1 - T2	relative compliment (asymmetrical)
T1 ! T2	absolute complement

The section “Query Language Examples” contains example queries using the various operators.

In addition to the query operators, the language contains qualifiers that can be used to limit which features within the tracks, are used in particular searches. For example, if a track contains features encoding all genes in a genome, the sequence limiter could be used to only examine features on a particular chromosome². Limiters are included to limit features by sequence, position on the sequence (ex. only features that occur within the x to y positions on the sequence), and length of the feature (ex. only features that are within x to y positions long).

Qualifiers:

T1 S:S1	only features on sequence “S1”
T1 <x, y>	only features with lengths from x to y
T1 <;a, b>	only features that are located on the sequence between a to b
T1 <x, y; a, b>	only features with lengths from x to y and that are located on the sequence between a to b
T1 {x, y}	only clusters that contain x to y overlapping features
T1 {x, y; a, b}	only clusters that contain x to y features, separated by a to b positions

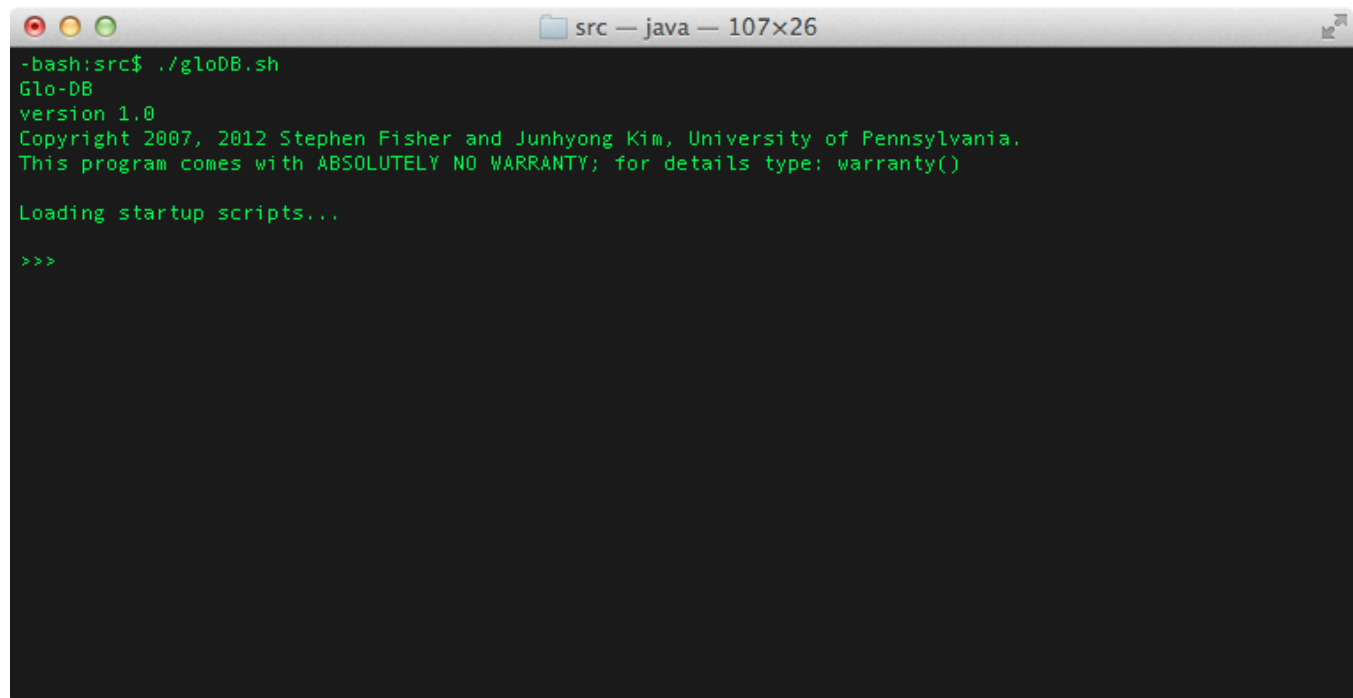
² Assuming a track named “gene_track” and the chromosome of interest named “xChrom”, the syntax would be “(T:gene_track S:xChrom)”. When combined with the previous example that found the overlap of genes and exons, this would be: T:exon_track AND T:gene_track S:xChrom

Command Line Interface:

The Command Line Interface (CLI) is based on the python scripting language and will accept most valid Python commands. The CLI can be used to load, modify, view, and save data. The CLI also provides access the internal data structures containing the track, feature, and sequence data. Thus, using Python commands, users can create their own scripts to automate their various tasks and even expand upon the built-in data operations. The use of scripts and user-defined data operations are discussed further in the section “Scripting Language” below.

When the program starts it automatically loads the file called ‘startup.py,’ containing various scripts included to facilitate the use of common functions. The user can modify this file to include their own scripts or to load addition files at startup.

By default, Glo-DB prints all CLI output in the History window of the graphical interface.

A screenshot of a terminal window titled 'src — java — 107x26'. The terminal shows the execution of the command './gloDB.sh'. The output includes the program name 'Glo-DB', version '1.0', copyright information for Stephen Fisher and Junhyong Kim at the University of Pennsylvania, a disclaimer 'ABSOLUTELY NO WARRANTY', and the message 'Loading startup scripts...'. The prompt '>>>' is visible at the bottom.

```
-bash:src$ ./gloDB.sh
Glo-DB
version 1.0
Copyright 2007, 2012 Stephen Fisher and Junhyong Kim, University of Pennsylvania.
This program comes with ABSOLUTELY NO WARRANTY; for details type: warranty()

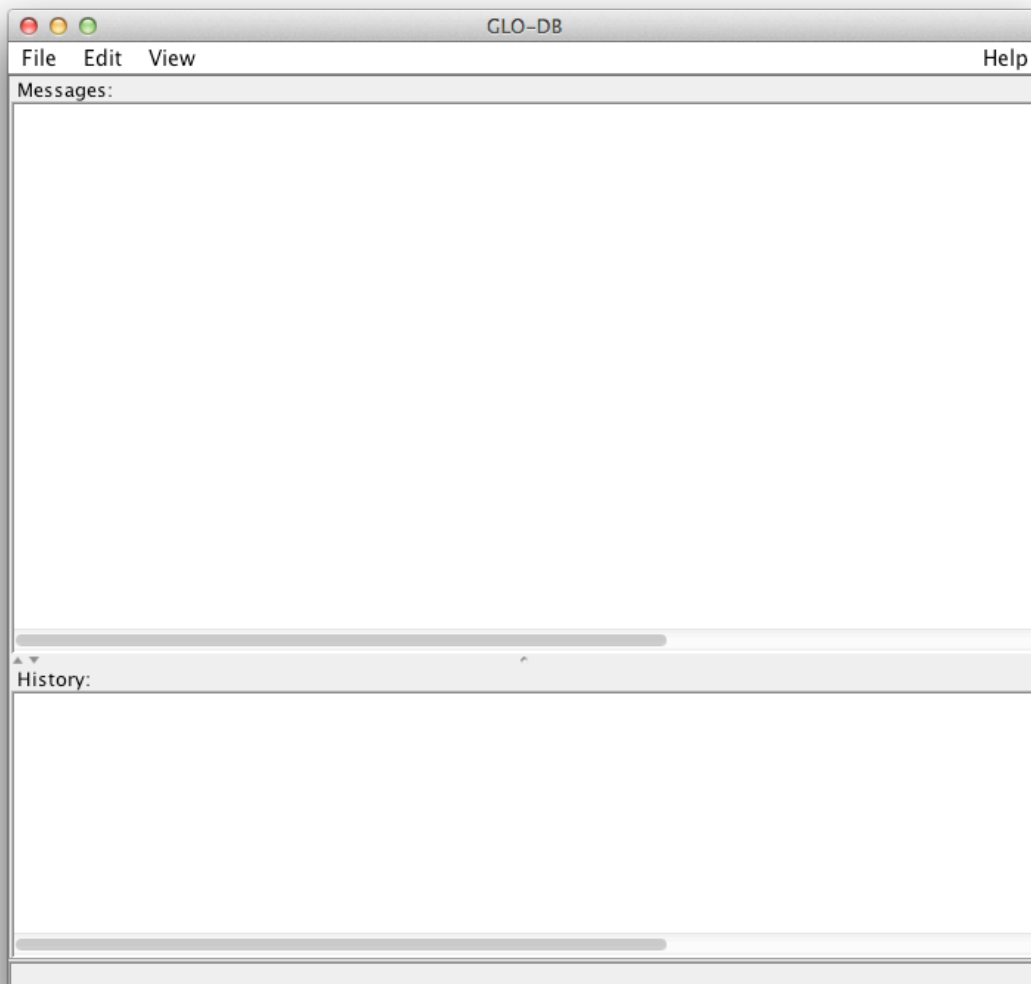
Loading startup scripts...

>>>
```

Graphical User Interface:

The Graphical User Interface (GUI) can be used to perform many of the built-in operations. The GUI has a main window which contains various menus, a ‘Messages’ area and a ‘History’ area. The ‘Messages’ window will display all status, warning, and error messages. The user can disable this window via the Edit menu, in which case all output will be displayed in the CLI. The verbosity of the feedback (status, warning, and error) can be set with the “setVerbose()” command.

The ‘History’ window contains the CLI equivalent of all commands initiated via the GUI. Thus, the History window will contain a log of all GUI derived commands. The user can copy and paste the commands in the History window directly into the CLI or into a file to be loaded into the CLI. Future versions of the GUI will include the ability to directly load or save command histories, and convert sequences of commands from the history into macros accessible through the CLI.



Menu Options:

File:

- Load Track (*.fas | *.fasta, *.gff)
- Save Track (*.fas | *.fasta, *.gff)
-
- New Sequence
- Load Sequence (*.fas | *.fasta)
-
- Quit

Edit:

- Copy History
- Clear History
- Select All History
-
- Enable Message Window
- Clear Messages
-
- Search Track

View:

- Browse Tracks
- Browse Sequences
-
- Display Track...
- Display Sequence...

Help:

- Help Topics
- API documentation
- Parser definitions
-
- About

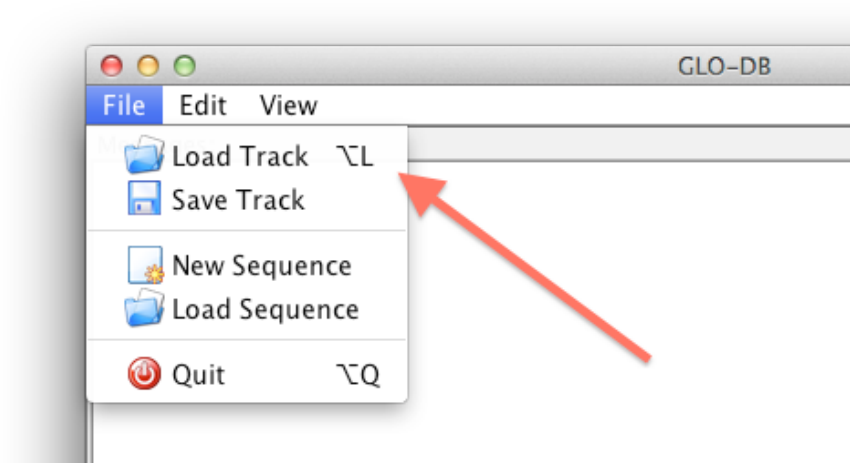
Usage Examples:

These examples use the chromosome chrX, in the human genome. The data files were downloaded from the UCSC genome browser (build hg18) and are located in the data folder, included with the installation of this application.

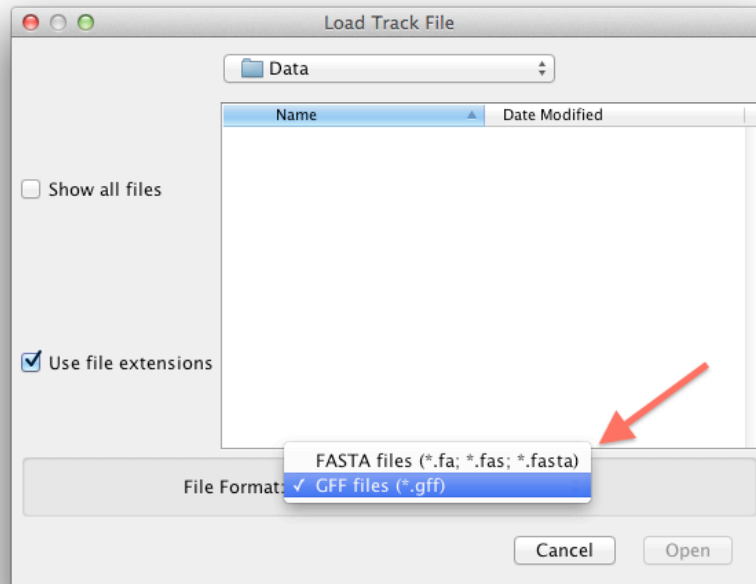
Working with the GUI: Compute SNPs located in conserved elements

Background: *SNPs* are single nucleotides that differ between two otherwise similar genomic sequences (eg DNA fragments). A *conserved element* is a nucleotide sequence (eg DNA fragment) that is similar across species or molecules (eg genes) within a single species. Hence this example explores the relationship between a set of single nucleotide positions that differ between a set of DNA fragments that are found across different species. This sort of query might be used to explore mutations in a set of DNA fragments that are found across different species.

1. Load SNPs track. The SNPs are located in the GFF formatted data file “snps.gff.” Using the GUI, select ‘Load Track’ from the ‘File’ menu or press ‘ALT-L’.



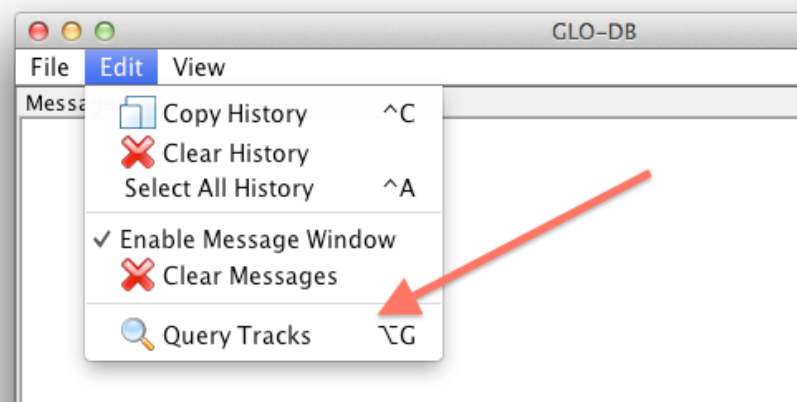
2. An ‘Open File’ dialog box will appear. In this dialog select ‘GFF files.’ Then select the file ‘snps.gff’ from the data directory and press the ‘Open’ button. This might take a minute or so to load due to the size of the file.



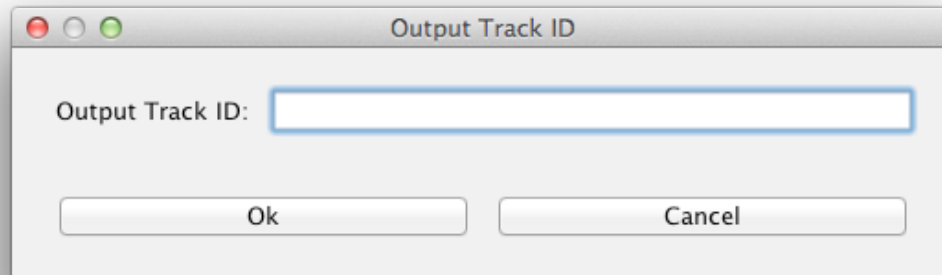
3. The 'History' window will display the command line equivalent to the GUI command we just ran.

```
>>> loadTrack("/Users/safisher/gloDB/./data/snps.gff", 3)
```

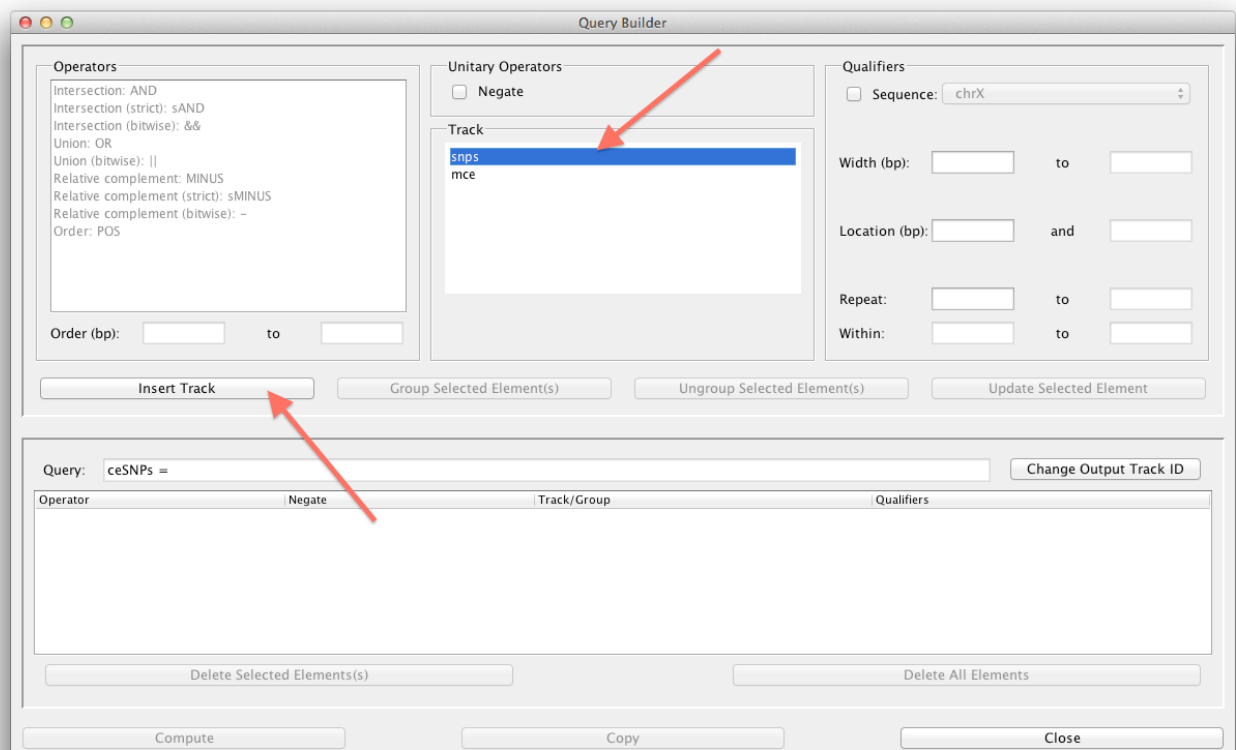
4. Load the track containing conserved elements ('mce.gff'). Using the GUI, select 'Load Track' from the 'File' menu and in the 'Load Track File' dialog select the 'mce.gff' file from the data folder.
5. Having loaded the SNPs and conserved elements, the user can query the tracks to create a new track that contains all SNPs that are located in conserved elements. Using the GUI, select 'Query Tracks' from the 'Edit' menu.



- A dialog box will appear asking for an id value for the output track; that is, the track which will contain the output from the query. In our case this track will contain all SNPs that are located in conserved elements, so let's call it 'ceSNPs'. Enter 'ceSNPs' in the id field and then press the 'Ok' button to continue.



- The Query Builder now appears. All of the query operations can be accessed from this dialog. To perform the query we are interested in here, select the 'snps' track and press the 'Insert Track' button.



- The track 'snps' has now been added to the query, enabling us to select an operator. Select the 'AND' operator and the 'mce' track. Then press the 'Insert Track' button. This will create the expression 'snps AND mcs' which will return a track containing all SNPs and conserved elements that overlap.

Query:

Operator	Negate	Track/Group
AND		snps mce

- We now need to remove all 'mce' features from the output track, since we want the output to contain the SNPs located in the conserved elements, and not the actual conserved elements. First we need to group the 'snps AND mce' operation. To do this select the two elements from the table in the lower pane (hold down the SHIFT key while using the mouse to select the items) and then press the 'Group Selected Element(s)' button.

Query Builder

Operators

- Intersection: AND
- Intersection (strict): sAND
- Intersection (bitwise): &&
- Union: OR
- Union (bitwise): ||
- Relative complement: MINUS
- Relative complement (strict): sMINUS
- Relative complement (bitwise): -
- Order: POS

Order (bp): to

Unitary Operators

☐ Negate

Track

snps
mce

Qualifiers

☐ Sequence:

Width (bp): to

Location (bp): and

Repeat: to

Within: to

Insert Track Group Selected Element(s) Ungroup Selected Element(s) Update Selected Element

Query:

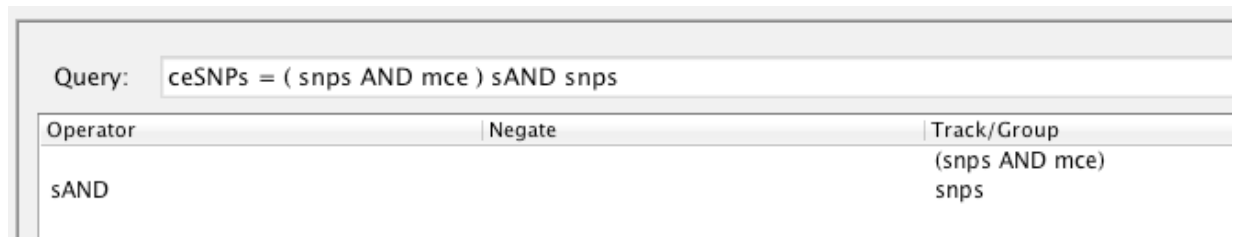
Operator	Negate	Track/Group	Qualifiers
AND		snps mce	

Delete Selected Element(s) Delete All Elements

Compute Copy Close

- Parenthesis are now placed around the 'snps AND mce' expression allowing us to perform an operation on this expression. Now select the 'snps' track and the 'sAND' operator (the 'strict'

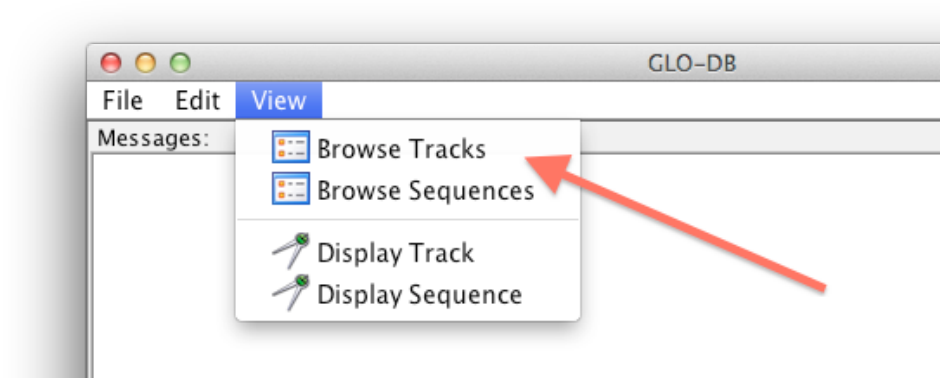
intersection), then press the ‘Insert Track’ button. The strict intersection will only include features in the output that exactly overlap; that is, features must have the same start and stop positions to be considered intersecting. This is not the case for the non-strict intersection (ie ‘AND’). Non-strict intersections only require that a single position overlap between two features for them to be considered intersecting. After pressing the ‘Insert Track’ button, the query will be updated, showing our completed expression.



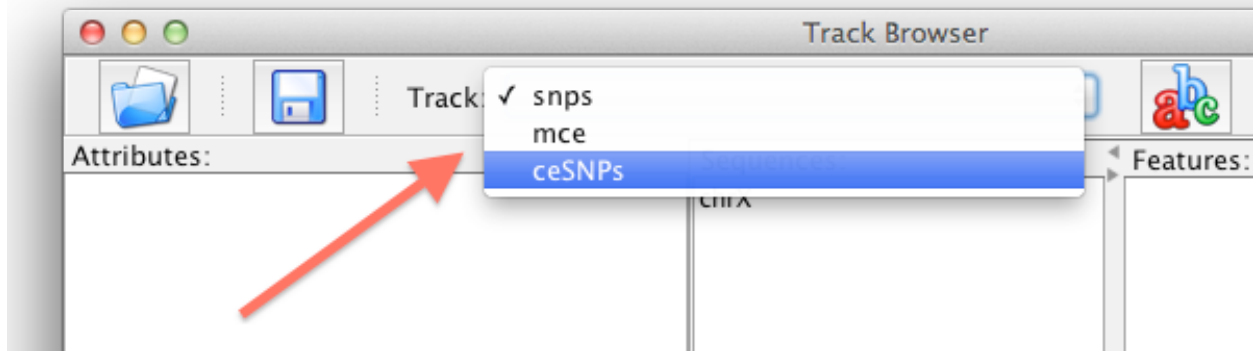
- The ‘Copy’ button will copy this expression to the clipboard for pasting into a text editor (eg to be included in a user defined function). For this tutorial we just want to run the expression, so press the ‘Compute’ button. Depending on the speed of your computer, the query might take a minute or so to complete. When the query has completed, press the ‘Close’ button to close the Query Builder dialog.



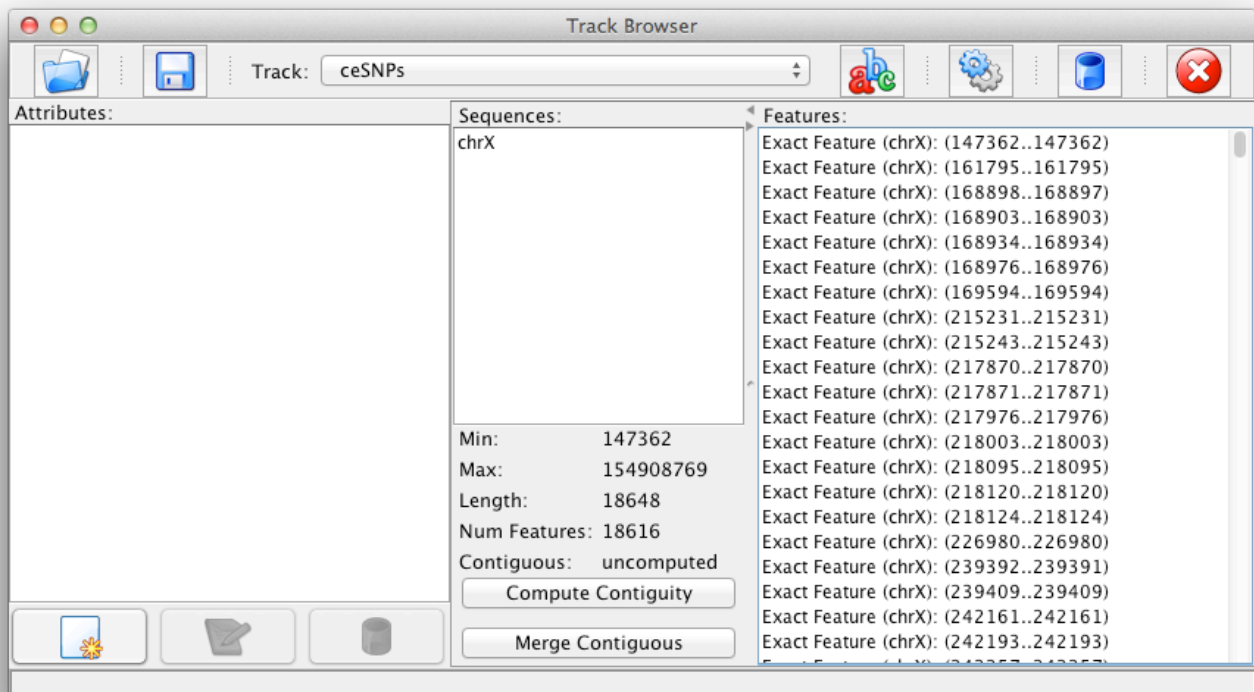
- We can use the Track Browser to view information about the various tracks that are loaded. To open the Track Browser select ‘Browse Tracks’ from the ‘View’ menu.



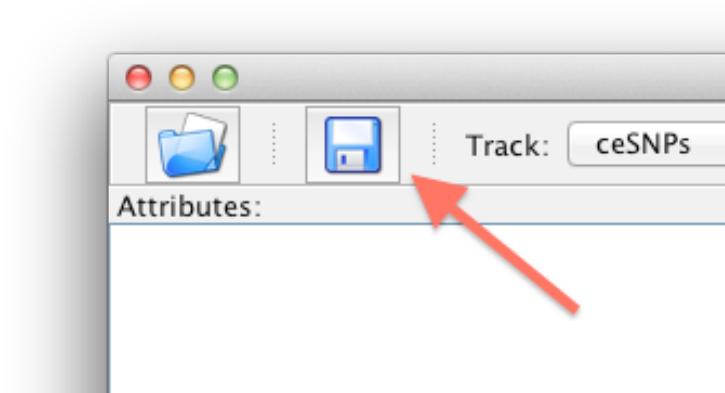
- At the top of the Track Browser select the ‘Track’ pull-down menu and select the track ‘ceSNPs’. This will cause the ‘ceSNPs’ track to be displayed.



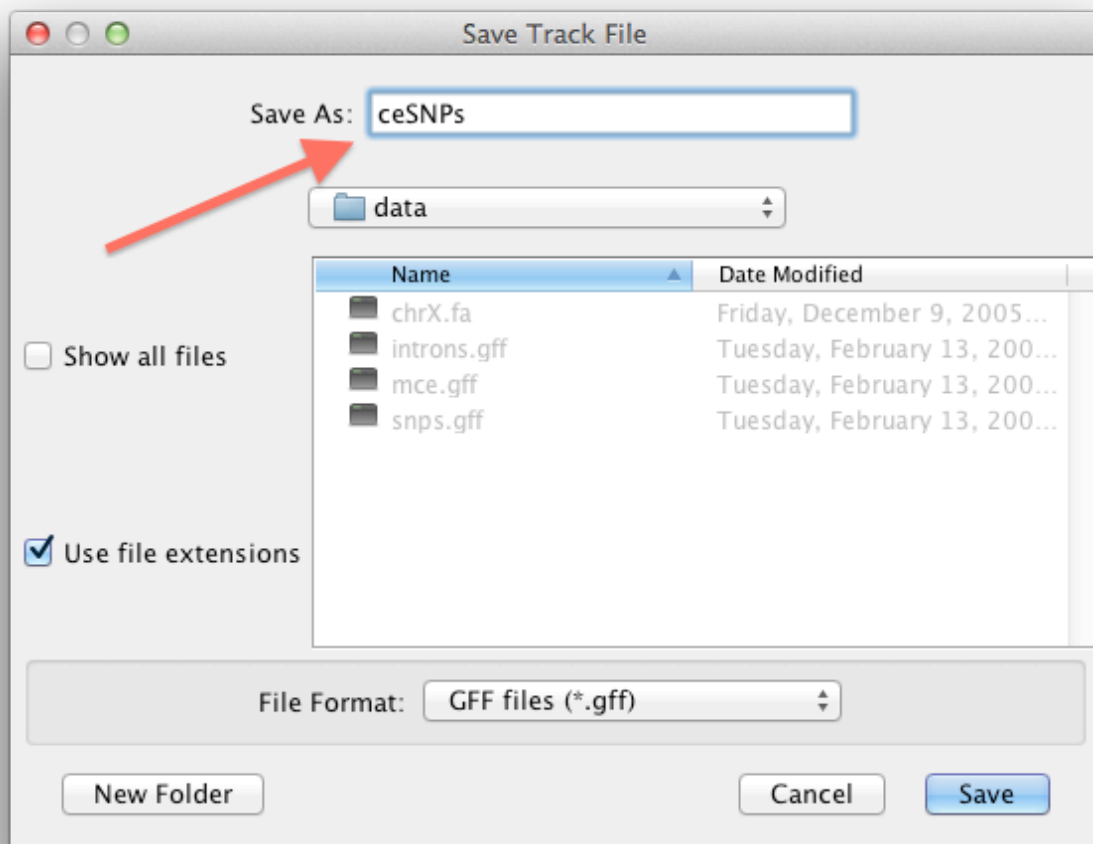
14. The Track Browser contains various panels with information about the selected track. The 'Attributes' panel is currently blank because there are no attributes associated with this track. The buttons at the bottom of the attributes panel can be used to add, change, or delete attributes. The 'Features' panel contains a list of all features in the selected track (features are not displayed for track containing more than 40,000 features). Double-clicking on a feature will bring up a dialog box that displays the attributes for that feature. Each feature only exists in relation to a sequence. The center panel lists the sequences that underlie the features in the specified track. This panel also includes various details about the track.



15. To save the track as a GFF file, select the 'Save' button in the Track Browser.



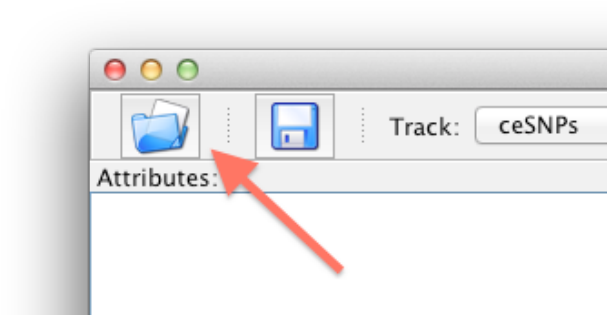
16. In the 'Save Track File' dialog box, enter the file name 'ceSNPs' and be sure that 'GFF files' is specified in the 'Files of type' pull-down menu. Then press the 'Save' button to create the output file.



Going between the GUI and CLI: Compute SNPs that are located in conserved elements or transcription factor binding sites

Background: Transcription factor binding sites are locations on DNA sequences where proteins bind to control the transcription of the DNA sequence (ie converting the DNA into RNA). So in this example we are looking at a more broad set of SNPs.

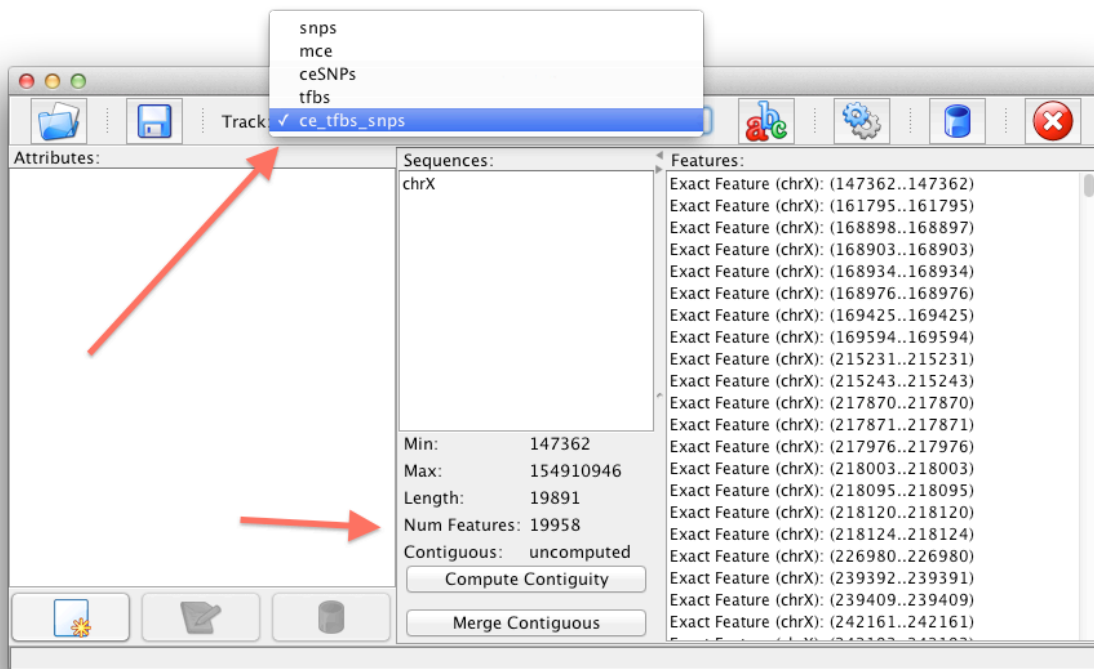
1. First we need to load the transcription factor binding sites (TFBS). These are located in the 'tfbs.gff' file. These can be loaded using the 'Load' button in the Track Browser.



2. To perform the query we need to compute the SNPs that exist in the 'tfbs' track. We can compute this the same way we computed the SNPs in the conserved elements: (snps AND tfbs) sAND snps. However, here we are looking for the union of both tracks so the complete expression will be: "ceSNPs OR ((snps AND tfbs) sAND snps)". We could create this expression in the Query Builder or we could enter this expression at the command line using the following command. This command creates a new track called 'ce_tfbs_snps' that contains all SNPs which are located in a conserved element or TFBS.

```
>>> compute("ce_tfbs_snps = ceSNPs OR ((snps AND tfbs) sAND snps)")
```

3. Once the command is complete we can see in the Track Browser that there were 19,958 features located in either conserved elements or TFBS.



- Using the 'Save' button in the Track Browser we can now save the output as a GFF file. We could also save the file by entering the following command in the command line. The command specifies that the track called 'ce_tfbs_snps' should be saved as a 'GFF' file into the file 'data/ce_tfbs_snps.gff'. Note that the '.gff' extension is optional and will be added to the filename if not included. The '0' specifies that the command should not overwrite a pre-existing file. If the user does want to overwrite an existing file, then use '1' instead of '0'.

```
>>> saveTrack("ce_tfbs_snps", GFF, "data/ce_tfbs_snps.gff", 0)
```

Working with objects in the CLI: Compute SNPs located in splice sites

- In this example we denote the position of splice sites as the 10 bps surrounding intron start and stop positions. The GFF file containing the introns we will be using, is called 'introns.gff'. We can load this file with the GUI as previously described. This file can also be loaded from the command line using the following command.

```
>>> loadTrack("data/introns.gff", GFF)
```

- The query we need to run is a little more complicated than the previous queries. The query will use the 'POS' command which returns features based on their order. First we will find all SNPs that are within 10 bps of the stop position of each intron (ie from 10 bps before each stop position to 10 bps after the stop positions). To do this we will use the 'POS' range of '-10 to 10'. The command to find the SNPs within 10 bps of the intron stop positions is: "introns POS{-10, 10} snps". While this will return the SNPs of interest, it will also return the associated introns; that is, the introns which containing SNPs within 10 bps of their stop positions. Thus when a match is found (ie a SNP is found with 10 bps of a intron), both the intron and SNP are included in the output track. We will deal with removing the introns shortly.

First, let's find the SNPs that are within 10 bps of the start of each intron. To do this we can just reverse the previous query as such: "snps POS{-10, 10} introns". The union of these two queries will thus contain a track with the relevant SNPs and the associated introns. To remove the introns from the output we need to add "sAND snps" to the query, as we did in the previous queries. The final query is shown below. This query can also be generated in the Query Builder.

```
>>> compute("spliceSNPs = ((introns POS{-10, 10} snps) OR (snps POS{-10, 10} introns))  
sAND snps")
```

3. To save the output as a GFF file we can use the following command.

```
>>> saveTrack("spliceSNPs", GFF, "data/spliceSNPs", 0)
```

4. The 'compute()' command returns a reference to the track object created by the query (ie 'spliceSNPs'). This reference can be assigned to a variable to allow for direct access to the track object, from the command line. Had we used the following compute command, we would have stored a reference to the track in the variable 'spliceSNPs'.

```
>>> spliceSNPs = compute("spliceSNPs = ((introns POS{-10, 10} snps) OR (snps POS{-10,  
10} introns)) sAND snps")
```

Since we didn't assign the reference, we can get a copy of the reference now using the 'getTrack()' command. When we ran the query we called the track object 'spliceSNPs' so we can now use that ID to get a reference to the track object. The following command will create a variable called 'spliceSNPs' that is holding a reference to a track object which has an ID of 'spliceSNPs'. The variable does not have to have the same name as the track ID, it's just often convenient.

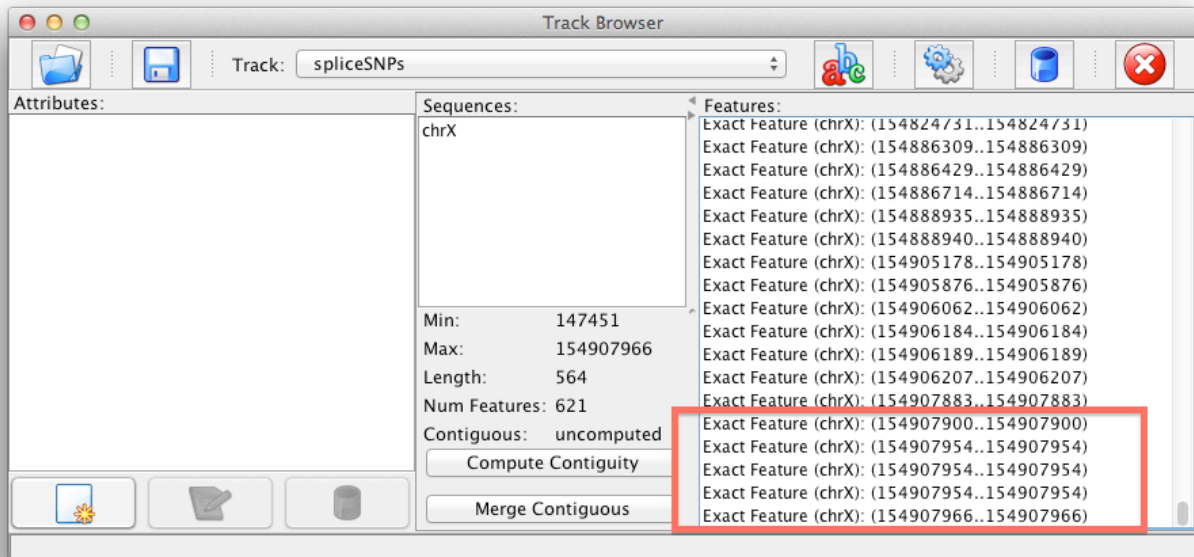
```
>>> spliceSNPs = getTrack("spliceSNPs")
```

5. With a reference to the spliceSNPs track object, we can now access the track object's Java methods (see "Class Methods" below). Methods exist to access the same track details as included in the Track Browser. For example, the following commands would return the number of features in the track and the track's length (cumulative number of bps of all features in the track). Note that the number of features '621' is more than the length of all features '564'. When computing the length of a track, overlapping base pairs are only counted once.

```
>>> spliceSNPs.numFeatures()  
621
```

```
>>> spliceSNPs.length()  
564
```

6. Looking at the features in the 'spliceSNPs' track in the Track Browser we see that there are duplicate features; that is, features with the same start and stop positions but different attribute values.



If we are unconcerned with the attribute values for each feature and only concerned with non-duplicate start and stop position, we could use the following command to remove the duplicate features.

```
>>> spliceSNPs = spliceSNPs.noRepeats ()
```

- Our variable 'spliceSNPs' now holds a reference to a new track object that is identical to the 'spliceSNPs' track but does not contain any duplicate features. Using the 'numFeatures()' method, we see that this new track only contains 619 features.

```
>>> spliceSNPs.numFeatures()
619
```

- Since the number of features is still greater than the length of the track (still 564 bps), it's likely there are SNPs larger than one base pair. We can use the length qualifier in the query language to limit our track to only features with a length of one base pair. However, to do this query we need to know the track's ID. Our original track had an ID of 'spliceSNPs'. We are now working with a new track that was created by the 'noRepeats()' method. Since every track object must have a unique ID, this new track has a different ID from our original track. To get the ID value for this new track we can use the 'getID()' method.

```
>>> spliceSNPs.getID()
'_spliceSNPs_3766077167719397885'
```

- When the 'noRepeats()' method ran it created a random ID for our new track. Since this ID isn't very descriptive, we can change it to something more useful.

```
>>> spliceSNPs.setID('spliceSNPsNR')
```

10. We can now use the following command to restrict our track to only include SNPs that are one base pair long. This command could also be run using the Query Builder.

```
>>> spliceSNPs = compute("spliceSNPsNR_1 = spliceSNPsNR<1>")
```

11. We now have a track that does not include any repeats and does not include any SNPs that are larger than one base pair. This new track object has an ID of 'spliceSNPsNR_1' and the variable 'spliceSNPs' is holding a reference to this track object. We can now review the number of features and length of this track. If using the Track Browser, you need to select the track by its ID value, which in this case is 'spliceSNPsNR_1'.

```
>>> spliceSNPs.numFeatures()
552
```

```
>>> spliceSNPs.length()
552
```

Working with Sequences: Extract nucleotide sequence for each transcription factor binding site containing SNPs

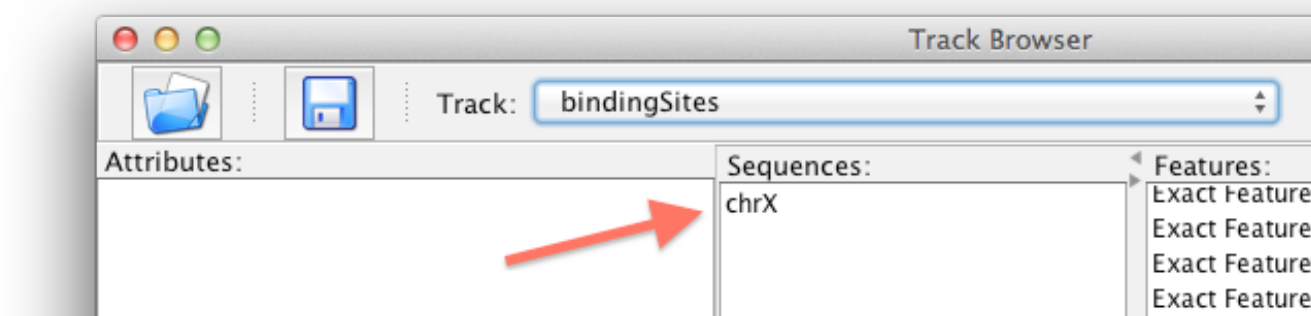
This example requires the hg18 human chromosome, chrX, sequence. The user can download this sequence, as a FASTA file, from the UCSC genome browser at:

<http://hgdownload.cse.ucsc.edu/goldenPath/hg18/chromosomes>

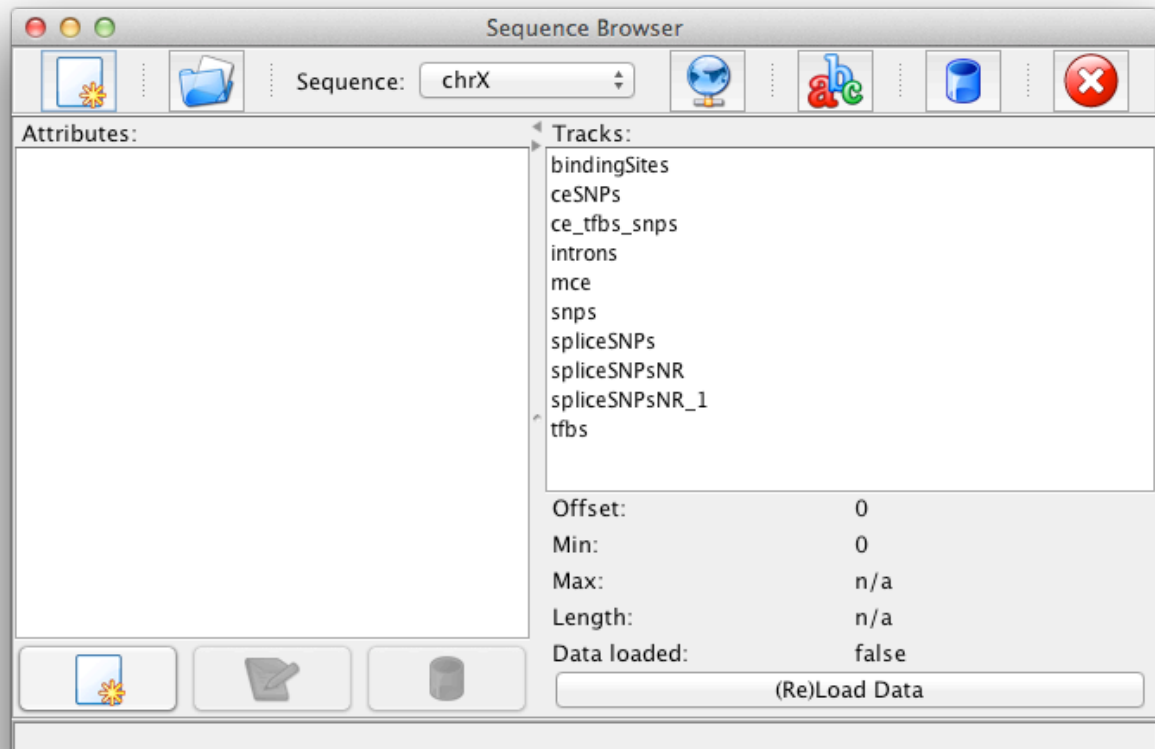
1. We previously found the SNPs located in transcription factor binding sites (TFBS). Now we want to find all TFBS that contain SNPs. We can do this in the same fashion.

```
>>> compute("bindingSites = (snps AND tfbs) sAND tfbs")
```

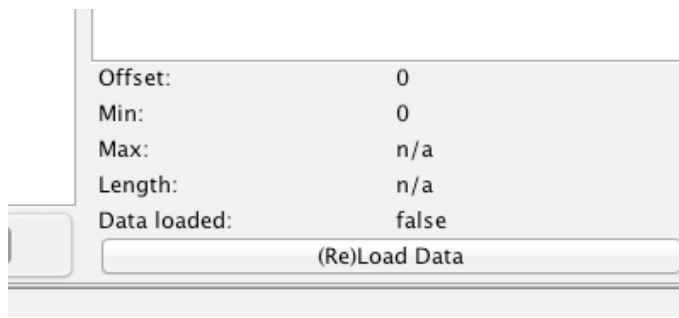
2. Now that we have a track containing the TFBS of interest, let's load the sequence data. When we loaded the first track containing features on the 'chrX' chromosome, a sequence object called 'chrX' was created. This sequence object was referenced by all future tracks containing features located on the 'chrX' chromosome. In the Track Browser we see that the new track we just created ('bindingSites') is also linked to 'chrX', as the binding sites in this track are located on the 'chrX' chromosome.



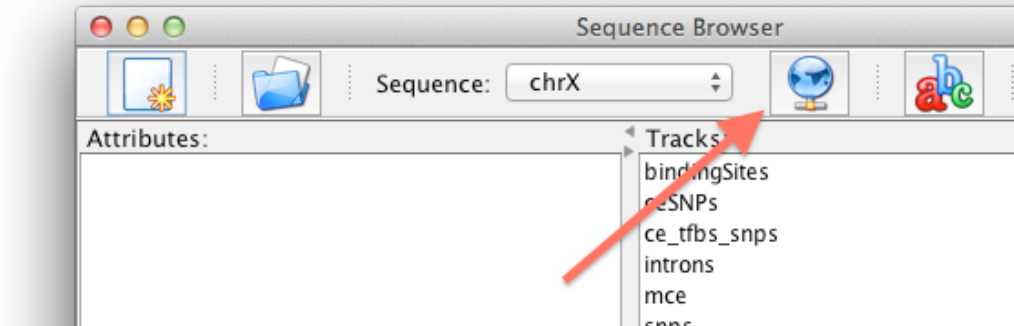
3. In the Track Browser, double click on the name of the sequence 'chrX' to open up the Sequence Browser.



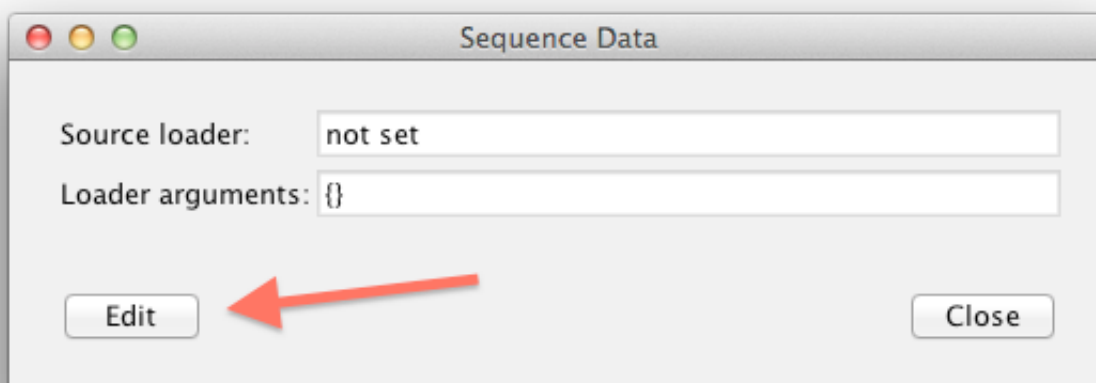
4. As with the Track Browser, the Sequence Browser displays information describing the selected sequence. Notice that the values for 'Max' and 'Length' are 'n/a'. These values are not set because we have not yet loaded the sequence data. The features we've been working with are locations on a sequence. They do not contain the sequence data and we didn't need the actual sequence data to work with the sequence locations. Thus the sequence details are unknown. GloDB created an empty sequence object called 'chrX' since that was necessary to use as a reference for our features. It is our responsibility to associate the sequence object with a data source.



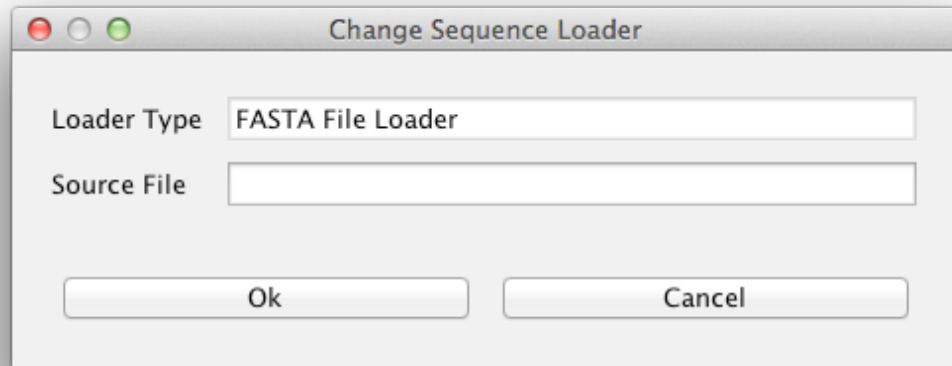
5. To associate the sequence object with a data source, select the source viewer button in the Sequence Browser. This will open up a 'Sequence Data' dialog box.



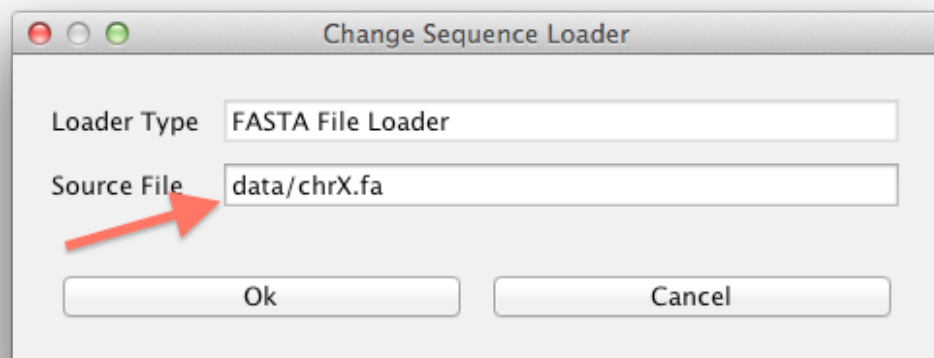
6. In the 'Sequence Data' dialog box, press the 'Edit' button, to change the source for the sequence data. This will open up a dialog box allowing for the setting of the data source file.



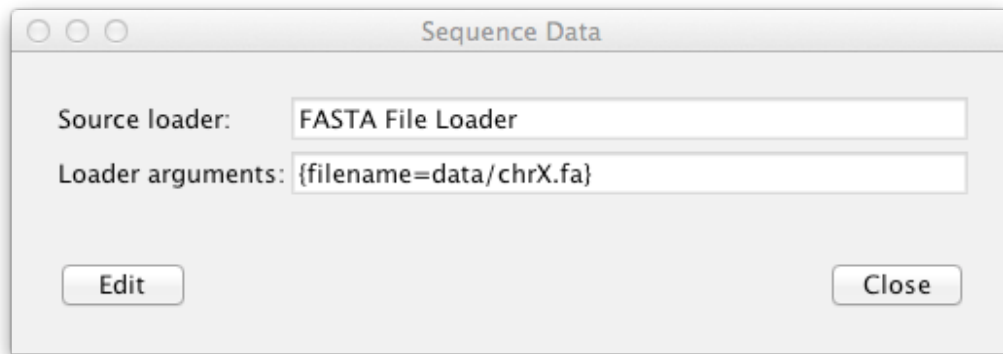
7. In the 'Change Sequence Loader' dialog the 'Loader Type' is not editable. Currently the application only allows sequence data to be loaded from FASTA files. Future versions will allow for the loading of sequence data from other sources.



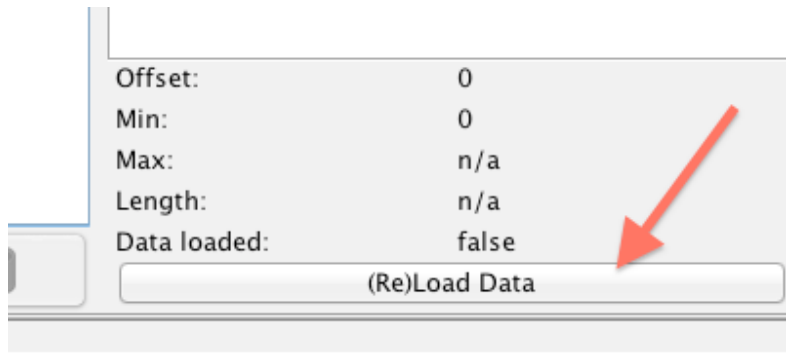
8. Enter the location of the FASTA file containing the chrX sequence data into the 'Source File' field. In our case the FASTA file is called 'chrX.fa' and is located in the data subdirectory. After entering the source file, press the 'Ok' button.



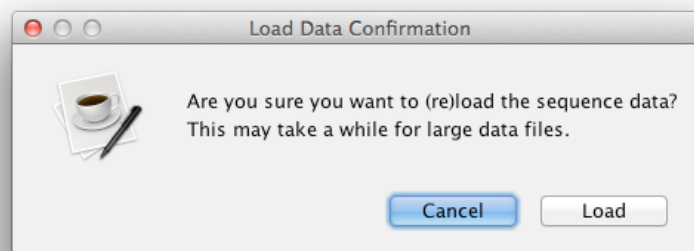
9. In the 'Sequence Data' dialog, the 'Loader arguments' should now contain a single argument which is 'filename' and the value of this argument should be the file you entered for the sequence source. In our case again this is the file called 'chrX.fa' located in the data subdirectory. If this is correct, then press the 'Close' button to continue.



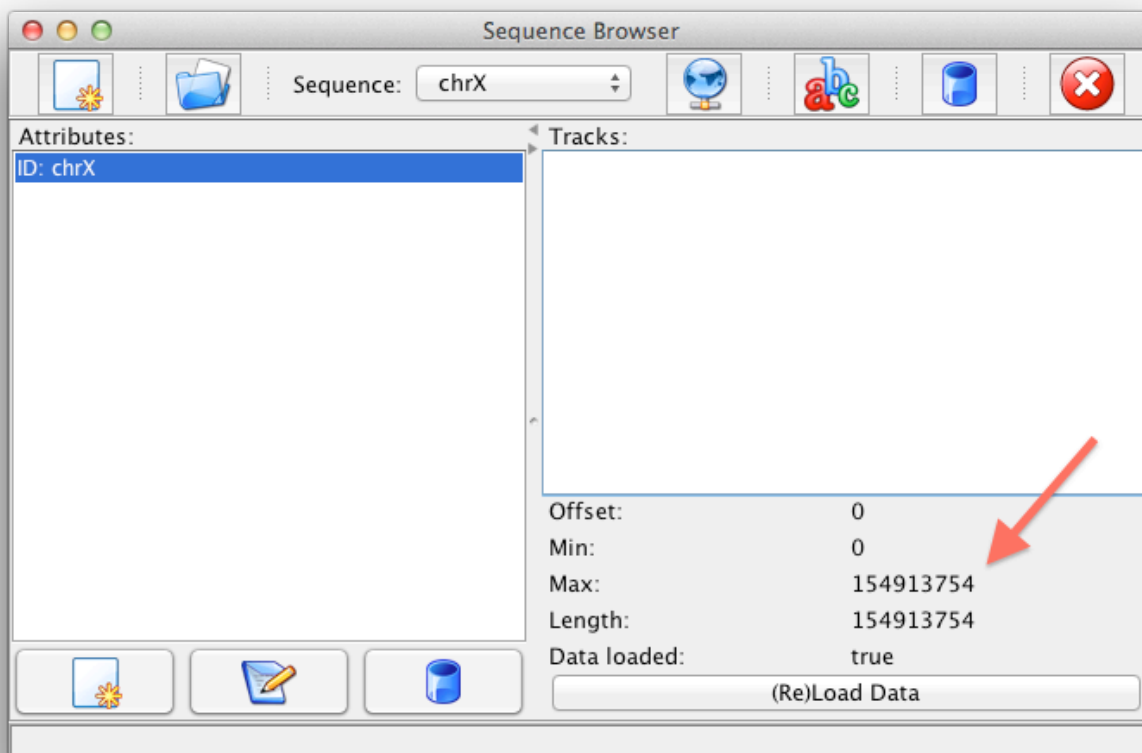
10. To load the data, press the '(Re)Load Data' button in the Sequence Browser.



11. A dialog will appear asking to confirm the loading of the data. Since this data file is 150 MB large, depending on the speed of the computer, it will take a significant amount of time to load this data file. To continue press the 'Load' button in the confirmation dialog box. While loading the data file, the GUI will become unresponsive, and feedback will be send to the CLI. The GUI will again become responsive when the data loading has completed.



12. After loading the data the Sequence Browser shows that the sequence contains 154,913,754 bps.



13. Now that the sequence data has been loaded, we can export the track containing binding sites as a FASTA file. The following command will save the track called 'bindingSites' as a FASTA file called 'bindingSites.fa' in the data subdirectory. The '0' means this command will not overwrite an existing file. If the '0' is changed to a '1', then this command would overwrite an existing file in the data directory called 'bindingSites.fa'.

```
>>> saveTrack("bindingSites", FASTA, "data/bindingSites.fa", 0)
```

Class Methods:

Track:

void addAttribute(String key, String value): Add an attribute.

void addFeature(Feature feature): Add a Feature to this Track.

void addFeatures(TreeSet features): Add all Features from the TreeSet. This is the same collection that is returned by `getFeatures()`, allowing users to easily copy all Features from one Track to another.

Object clone(): Returns a shallow copy of this Track.

boolean contains(Feature feature): Returns true if 'feature' exists in this Track.

boolean contains(String sourceID): Returns true if this Track contains any Features on 'sourceID.'

boolean containsAttribute(Object key): Returns true if attribute 'key' exists.

void delAttribute(Object key): Removes the attribute 'key.'

Iterator featureIterator(): Returns an Iterator over the set of Features in this Track.

TreeSet featuresBySource(String sourceID): Returns the set of Features in this Track that are all located on the sequence 'sourceID.'

Object getAttribute(Object key): Get the value for attribute 'key.'

HashMap getAttributes(): Get the attributes.

ArrayList getData(): Returns the sequence data. Sequence data that occurs on different contigs or is non-contiguous with separate items in the ArrayList.

String getDataFASTA(): Returns the sequence data formatted as a multi-sequence FASTA file.

String getDataFormatted(): Returns the sequence data with "\n" inserted every Sequence.FORMAT_WIDTH characters and a blank line will be inserted between gaps in the data.

TreeSet getFeatures(): Returns the entire set of Features in this Track, sorted by their start positions.

TreeSet getFeaturesByMax(): Returns the entire set of Features in this Track, sorted by their stop positions.

String getID(): Get the ID.

int getMax(): Returns the maximum stop position in the Track. Will return '-1' if there are no Features or if the Track contains Features on different contigs (ie isSingleSource() returns 'false').

int getMin(): Returns the minimum start position in the Track. Will return '-1' if there are no Features or if the Track contains Features on different contigs (ie isSingleSource() returns 'false').

Set getSourceSet(): Returns the set of source Sequence objects that underly the Features in this Track.

boolean isContiguous(): Returns 'true' if the Track does not contain gaps between Features. If the Features occur on different sequences , then this will return 'false'.

boolean isSingleSource(): Returns 'true' if the Features contained in the Track all refer to the same Sequence. This is similar to isContiguous() but allows for gaps between Features.

int length(): Returns the number of positions contained in the Track. Overlapping positions will only be counted once.

void mergeContiguous(): This will merge all overlapping Features in the Track, creating new Feature objects as necessary.

Track noRepeats(): Returns a copy of the track without any duplicate features (based on start/stop values).

int numFeatures(): Returns the number of Features contained in the Track. If Features exactly overlap, they will be still be counted separately.

int numSources(): Returns the number of Sources spanned by the Track.

boolean overlaps(Feature feature): Returns 'true' if the Feature 'feature' overlaps at least one Feature in this Track.

boolean overlaps(Track track): Returns 'true' if any Feature in 'track' overlaps a Feature in this Track.

void removeFeature(Feature feature): Adds 'feature' to this Track.

void setFeatures(TreeSet features): Replaces the existing set of Feature in the Track with the Features in 'features.'

String toString(): Returns list of Sequences and the number of Features per Sequence.

String toStringFull(): Returns list of Features including their respective start/stop positions.

Features:

void addAttribute(String key, String value): Add an attribute.

int compareTo(Object feature): Compares the order of 'feature' with this Feature. Returns '-1', '0', or '1' as this Feature is less than, equal to, or greater than the specified feature ('feature').

boolean contains(Feature feature): Returns true if the Feature 'feature' exists on the same Sequence and within the start/stop boundaries of this Feature.

int contains(int pos): Returns '-1' if this Feature exists after the integer 'pos', returns '0' if 'pos' is contained in this Feature, and '1' if 'pos' occurs after this Feature. This assumes 'pos' is positive and within this Feature's Sequence boundaries.

boolean containsAttribute(Object key): Returns true if attribute 'key' exists.

void delAttribute(Object key): Remove an attribute.

boolean equals(Object feature): Returns true if the Feature 'feature' has the same source and start/stop positions as this feature. Attributes are not considered for this comparison.

Object getAttribute(Object key): Get value for attribute 'key'.

HashMap getAttributesMap(): Get the attributes as a HashMap.

String getData(): Returns the underlying sequence data.

String getDataFormatted(): Returns the Sequence data, with "\n" inserted every Sequence.FORMAT_WIDTH characters (usually 50 to 80 chars).

int getMax(): Returns the maximum position of the Feature. If the Feature consists of a fuzzy Feature, this may not be equal to 'stop'.

int getMin(): Returns the initial position of the Feature. This should return the same value as `getStart()`.

Sequence getSource(): Returns the underlying Sequence object.

String getSourceID(): Returns the underlying Sequence object's ID.

int getStart(): Returns the start position of the Feature. This should return the same value as `getMin()`.

int getStop(): Returns the maximum position of the Feature. If the Feature consists of a fuzzy Feature, this may not be the maximum position.

int length(): Returns the number of positions contained in the Feature.

Feature overlap(Feature feature): Returns the overlapping region between 'feature' and this Feature. The returns the overlapping region as a new Feature.

boolean overlaps(Feature feature): Returns true if the Feature 'feature' has at least one position that overlaps positions in this Feature.

String toString(): Only returns the basic Feature information.

String toStringFull(): Returns all Feature information, except the data.

Sequence:

void addAttribute(Object key, Object value): Add a sequence attribute.

boolean contains(Feature feature): Returns 'true' if 'feature' is contained in this Sequence object. This will return 'false' if the Feature's source ID doesn't match this Sequence's ID.

boolean contains(int pos): Returns 'true' if the position 'pos' is contained in this Sequence object.

boolean containsAttribute(Object key): Returns 'true' if the attribute 'key' exists.

void delAttribute(Object key): Removes the attribute.

Object getAttribute(Object key): Get a sequence attribute.

HashMap getAttributes(): Get the sequence attributes.

String getData(): Returns the Sequence data as a single unformatted string.

String getDataBounded(int min, int max): Returns the sequence data between position '(min-1)' and position 'max'. Goes from ((min-1) to max) because java Strings go from (0 to (length-1)) and the actual position data assumes (1 to length)

String getDataBoundedFormatted(int min, int max): Returns the bounded sequence data with "\n" inserted every FORMAT_WIDTH characters.

String getDataFormatted(): Returns the sequence data with "\n" inserted every FORMAT_WIDTH characters.

String getID(): Get the ID.

int getMax(): Returns the maximum position of the Sequence on the chromosome. If the dataLoader isn't set and thus no data is loaded, then will return -1.

int getMin(): Returns the initial position of the Sequence on the chromosome. This will return the same value as getOffset().

int getOffset(): Returns the Sequence starting position on the chromosome.

boolean isDataLoaded(): Returns true if data was loaded.

int length(): Returns the length of the data string. If the dataLoader isn't set and thus no data is loaded, then will return -1.

void setAttributes(HashMap attributes): Set the sequence attributes.

void setData(String data): Set the Sequence data, expecting a single unformatted string. This will set the dataLoaded flag to 'true'.

void setOffset(int offset): Set the Sequence starting position on the chromosome.

String toString(): Returns attributes information.

Query Language Examples:

Set Based Operators:

```
Sequence: |-----|
           |012345678901234567890123456789|
Track T1:   r-s   t-----u v---w   x-y
Track T2:   a-b   c-d e-f g---h i-j

T1: { (r,s), (t,u), (v,w), (x,y) }
T2: { (a,b), (c,d), (e,f), (g,h), (i,j) }
```

A OR B **union** (symmetrical). This function returns a track containing all features from tracks A and B. Features that are identical in both tracks will only be included once.

Example: **T1 OR T2** => {(r,s), (t,u), (v,w), (x,y), (c,d), (e,f), (g,h), (i,j)}

A AND B **intersection** (symmetrical). This function returns a track containing all features in A that overlap with a feature in B and similarly all features in B that overlap with a feature in A.

Example: **T1 AND T2** => {(r,s), (t,u), (v,w), (c,d), (e,f), (g,h), (i,j)}

A sAND B **intersection** (symmetrical). This function returns a track that contains features in A that exactly overlap with a feature in B; that is, the feature in A must have the same start and stop positions as the feature in B.

Example: **T1 sAND T2** => { (r, s) }

A MINUS B **relative compliment** (asymmetrical). This function returns a track containing any features in A that do not overlap with a feature in B.

Example: **T1 MINUS T2** => {(x,y)}

Example: **T2 MINUS T1** => {}

A sMINUS B **relative compliment** (asymmetrical). This function returns a track containing any features in A that do not exactly overlap with a feature in B; that is, a feature in A is only included if it does not have the same start and stop positions as a feature in B.

Example: **T1 sMINUS T2** => {(t,u), (v,w), (x,y)}

Example: **T2 sMINUS T1** => {(c,d), (e,f), (g,h), (i,j)}

A POS{X, Y} B **order** (asymmetrical). This function returns a track containing any features in A that are followed X to Y base pair positions by a feature in B. The relevant features in B are also included in the output track.

Example: **T1 POS{5} T2** => {(t,u), (i,j)}

Example: **T2 POS{5} T1** => {(a,b), (t,u), (e,f), (v,w)}

Example: **T1 POS{-5} T2** => {(t,u), (e,f)}

Example: **T1 POS{3,6} T2** => {(r,s), (c,d), (t,u), (i,j)}

Example: **T1 POS{-5,-1} T2** => {(r,s), (t,u), (e,f), (g,h), (v,w), (i,j)}

Binary Based Operators:

```
Sequence: |-----|
           |012345678901234567890123456789|
Track T1:   r-s     t-----u v---w   x-y
Track T2:   a-b     c-d e-f g---h i-j

T1: { (1,3), (8,16), (18,22), (25,27) }
T2: { (1,3), (7,9), (11,13), (15,19), (21,23) }
```

- A || B** **union** (symmetrical). This function returns a track containing all features from tracks A and B. Any features that overlap will be merged into a single (new) feature that spans the entire length of the overlapping features.
Example: **T1 || T2** => {(1,3), (7,23), (25,27)}
- A && B** **intersection** (symmetrical). This function returns a track containing all base pair positions that overlap between tracks A and B. New features are created, as necessary, to encode the portions of the features that overlap.
Example: **T1 && T2** => {(1,3), (8,9), (11,13), (15,16), (18,19), (21,22)}
- A – B** **relative compliment** (asymmetrical). This function returns a track containing the portions of the features in A that do not overlap with features in B. New features are created, when needed to encode portions of features in A.
Example: **T1 – T2** => {(10,10), (14,14), (20,20), (25,27)}
Example: **T2 – T1** => {(7,7), (17,17), (23,23)}
- ! A** **absolute complement**. This function returns a track containing features which encode all base pair positions not encoded by the features in the track A.
Example: **! T1** => {(0,0), (4,7), (17,17), (23,24), (28,29)}
Example: **! T2** => {(0,0), (4,6), (10,10), (14,14), (20,20), (24,29)}

Additional Examples:

```
t1 sMINUS t2 AND t2:      {(t,u), (v,w), (c,d), (e,f), (g,h), (i,j)}
t1 sMINUS (t2 AND t2):   {(t,u), (v,w), (x,y)}
(t1 MINUS t2) OR (t2 MINUS t1): {(x,y)}
(t1 sMINUS t2) OR (t2 sMINUS t1): {(t,u), (v,w), (x,y), (c,d), (e,f), (g,h), (i,j)}
! t1 && t2:               {(7,7), (17,17), (23,23)}
t1 && ! t2:               {(10,10), (14,14), (20,20), (25,27)}
! (t1 && t2):             {(0,0), (4,7), (10,10), (14,14), (17,17), (20,20), (23,29)}
! (t1 || t2):             {(0,0), (4,6), (24,24), (28,29)}
! (t1 - t2 && t1):        {(0,9), (11,13), (15,19), (21,24), (28,29)}
! (t1 - (t2 && t1)):      {(0,9), (11,13), (15,19), (21,24), (28,29)}
(t1 - t2) || (t2 - t1):   {(7,7), (10,10), (14,14), (17,17), (20,20), (23,23), (25,27)}
```


Qualifiers:

```

Sequence M:  |-----|
              |012345678901234567890123|
Track T1:    n-----o p-q

Sequence N:  |-----|
              |012345678901234567890123456789|
Track T1:    r-s      t-----u v---w   x-y
Track T2:    a-b      c-d e-f g---h i-j

T1: { (n,o) , (p,q) , (r,s) , (t,u) , (v,w) , (x,y) }
T2: { (a,b) , (c,d) , (e,f) , (g,h) , (i,j) }

```

A S:S1

sequence. This function returns a track containing any features in A that are located on sequence “S1”.

Example: **T1 S:SM** => {(n,o), (p,q)}

Example: **T1 S:SN** => {(r,s), (t,u), (v,w), (x,y)}

Example: **(T1 OR T2) S:M** => {(n,o), (p,q)}

Example: **(T1 OR T2) S:N** => {(r,s), (t,u), (v,w), (x,y), (c,d), (e,f), (g,h), (i,j)}

A <x, y>

length. This function returns a track containing any features in A that have lengths from x to y.

Example: **T1 <3>** => {(p,q), (r,s), (x,y)}

Example: **T1 <2,6>** => {(p,q), (r,s), (v,w), (x,y)}

Example: **(T1 OR T2) <4,6>** => {(v,w), (g,h)}

A <;a, b>

location. This function returns a track containing any features in A that are located on a sequence between base pair positions a and b (inclusive).

Example: **T1 <;10>** => {(p,q), (v,w), (x,y)}

Example: **T1 <;8,17>** => {(p,q), (t,u)}

Example: **(T1 OR T2) <;20,30>** => {(x,y), (i,j)}

A <x, y; a, b>

length and location. This function returns a track containing any features in A that have lengths from x to y and are located on a sequence between a to b.

Example: **(T1 OR T2) <4,100;6,20>** => {(t,u), (g,h)}

A {x, y; a, b}

clustered features. This function returns a track containing groups of x to y features from A that are each a to b positions away from one another. If a and b are absent, the features that overlap are grouped together.

Example: **T1 {2;1,3}** => {(n,o), (p,q), (t,u), (v,w)}

Example: **(T1 OR T2) {4}** => {(c,d), (t,u), (e,f), (g,h)}

Example: **(T1 OR T2) {2}** => {(c,d), (t,u), (g,h), (v,w)}*

* Note that (e,f) is not included. (c,d) is matched with (t,u). (e,f) does not match (g,h), so it is skipped. (g,h) matches (v,w).

Scripting Language:

The command line interface (CLI) contains a scripting language (Jython) that is based on Python. The interface allows for most Python commands and syntax. For documentation on Python and Jython see:

- <http://www.jython.org>
- <http://www.python.org>

Users should review the script files included with the installation to view examples of how to access built-in objects from within the CLI.

startup.py: This file is loaded when the application begins. It contains various functions that facilitate interactions between the built-in objects and the CLI.

header.py: This file contains standard definitions related to GLO-DB and should be included in any user defined modules.

test.py: This module contains routines and objects used to test the application. Simple Tracks and Features are created and then used to perform basic operations which are then compared to the expected output.

testComp.py: This module was created to further test the computational algorithms in GLOB. Each of these functions will return a new Track which will not be included in the GLOB trackPool. These functions are external implementations of the internal algorithms and were created to test the correctness of the built-in algorithms (see `parser.Operations.java`). These functions were created with an emphasis on correctness rather than efficiency and speed.

cluster.py: This module was created as an example of how to manipulate Tracks and Features at the command prompt. This module will merge all Features in the Track that are within a specified distance of one another. New Features will be created to span the entire cluster. A threshold sets the minimum number of Features necessary to be considered a cluster and thus be included in the output set. A new Track will be returned, containing the clusters. If there are no matches, then 'null' will be returned.

Batch Mode:

Both gloDB.sh (Linux/Mac) and gloDB.bat (Windows) allow for the inclusion of up to *nine* command line arguments. The command line arguments are sent directly to the CLI at startup. When the “-batch” argument is given, GLO-DB will automatically quit after running the remaining arguments.

Example batch mode:

To run a file “ex_batch.py” which contains a series of GLO-DB commands:
gloDB.bat -batch “import ex_batch”

---- contents of file: ex_batch.py ---

This file will load a track containing exons (“exons.gff”) and one containing genes (“genes.gff”). The
intersection of these tracks will be computed and saved to a GFF file called “intersect.gff”

load standard definitions

from header import *

from startup import *

load tracks containing exons and genes

loadTrack("data/exons.gff",GFF)

loadTrack("data/genes.gff",GFF)

compute intersection of tracks and store output in new track called “_g”

compute("_g = exons and genes")

save track “_g” to GFF file called “intersect.gff”

if the file already exists, then it will be overwritten

saveTrack("_g", GFF, "intersect.gff", 1)

System Requirements:

The system requirements will depend on the size of the datasets being processed. The application has been tested to run on Mac OS 10.6, 10.7, and 10.8, RedHat Linux 5, and Windows XP.

Minimum:

- 128 MB of RAM is sufficient to run the application. Using Mac OS 10.8 the tutorial takes 1.5 GB of RAM.
- 15 MB of hard drive space for application files. Sample data files will require an additional 250 MB of hard drive space.
- Java version 1.5

Known Issues:

- Many features still not implemented in GUI.
- GenBank (*.gb) file format not implemented.
- Need better integration with Genome Browser. Upgrading from Swing to JavaFX will allow for display of Genome Browser webpage from within app (requires Javascript). Also should have export option that makes sure attributes field is properly formatted in GFF files for importing into Genome Browser (the first word in attributes field needs to be unique).
- Need to allow for loading of FASTA files that contain multiple sequences.
- Need to implement database and URL options for loading sequence data.
- Should include “contained()” and “xor()” examples in Jython.

Contact Information:

Dr. Stephen Fisher
Biology Department
University of Pennsylvania
433 S. University Ave
Philadelphia, PA 19104

safisher@sas.upenn.edu
(215) 898-8395

Dr. Junhyong Kim
Biology Department
University of Pennsylvania
433 S. University Ave
Philadelphia, PA 19104

junhyong@sas.upenn.edu
(215) 746-5187

Microsoft, Windows, is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Copyright © 2007, 2012
Stephen Fisher and Junhyong Kim
Biology Department, University of Pennsylvania
All Rights Reserved.