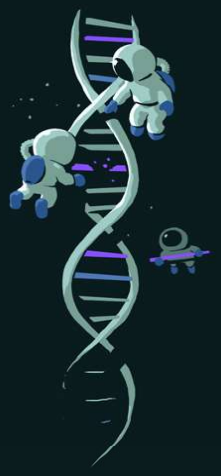
JIDA



Janeesh Kaur Bansal,

Isabel Rachel Thompson,

Diego Alejandro Pava Mejia,

Aravind Venkata Pattisapu

Contents

1. Introduction

1.1. What is JIDA?

JIDA is a web application that retrieves SNP information for a genomic region of interest in *Homo sapiens* chromosome 21 and calculates specific summary statistics for specified populations. Users can search by RS value, gene name or aliases and finally genomic positions. They can select the summary statistics of interest alongside which populations they would like data outputs for. Additionally, for the sliding window statistics calculations, the user can input their window size preference which would output the required results.

The summary statistics are returned in a table with the option of viewing graphs for statistics versus populations. Moreover, the user is presented with interactive graphs for each statistic, including the sliding window values against the genomic positions. The graphs allow the selection/deselection of populations for comparing results and can be downloaded as a PNG to present key findings. The summary statistics can be downloaded as a CSV file to store for later usage by the user.

The summary statistics provided by the website are nucleotide diversity, haplotype diversity, Tajima's D and FST. Information for the British, Finnish, Colombian, Punjabi and Telugu populations are present

in the database and is utilised by the web application. The website is simple to navigate through and produces results which can be utilised by users in developing biological research.

2. The Software

2.1. Navigating the website

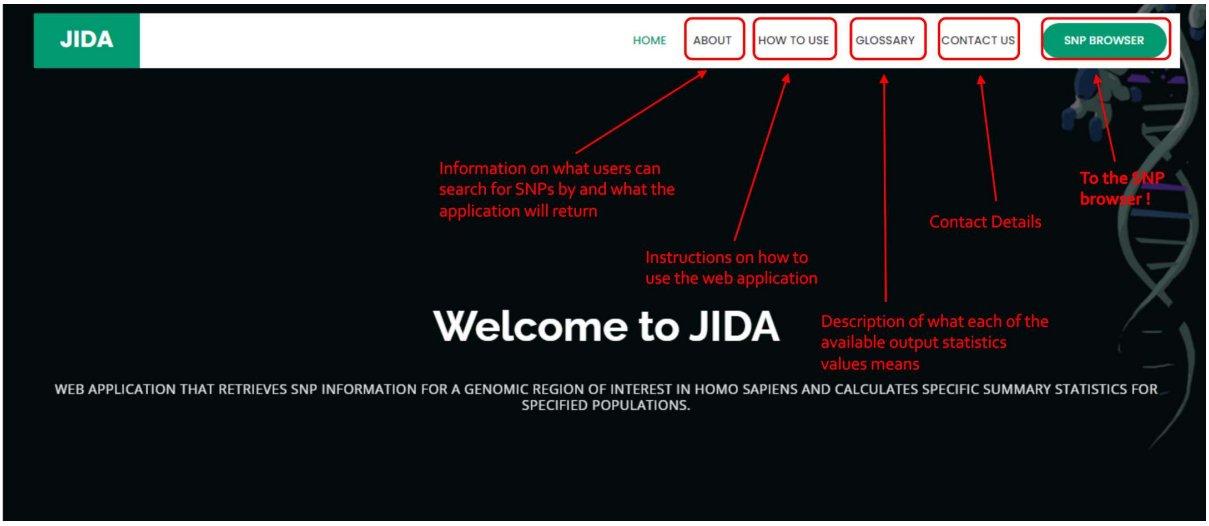


Figure 1: ***Annotated screenshot of the JIDA web application homepage.*** *The annotations describe what information can be found by clicking on each option of the navigation bar. The home/welcome page links all the sections of the website with a link to the SNP browser.*

Figure 1 shows the home page of the JIDA web application. Either by navigating to pages through the navigation bar (top, white in colour) or by simply scrolling down, the user can access supplementary information such as how to use the application, a glossary for the statistics used, and contact details. To access the SNP browser itself, the user should click the SNP BROWSER button on the navigation bar (top right, green in colour).

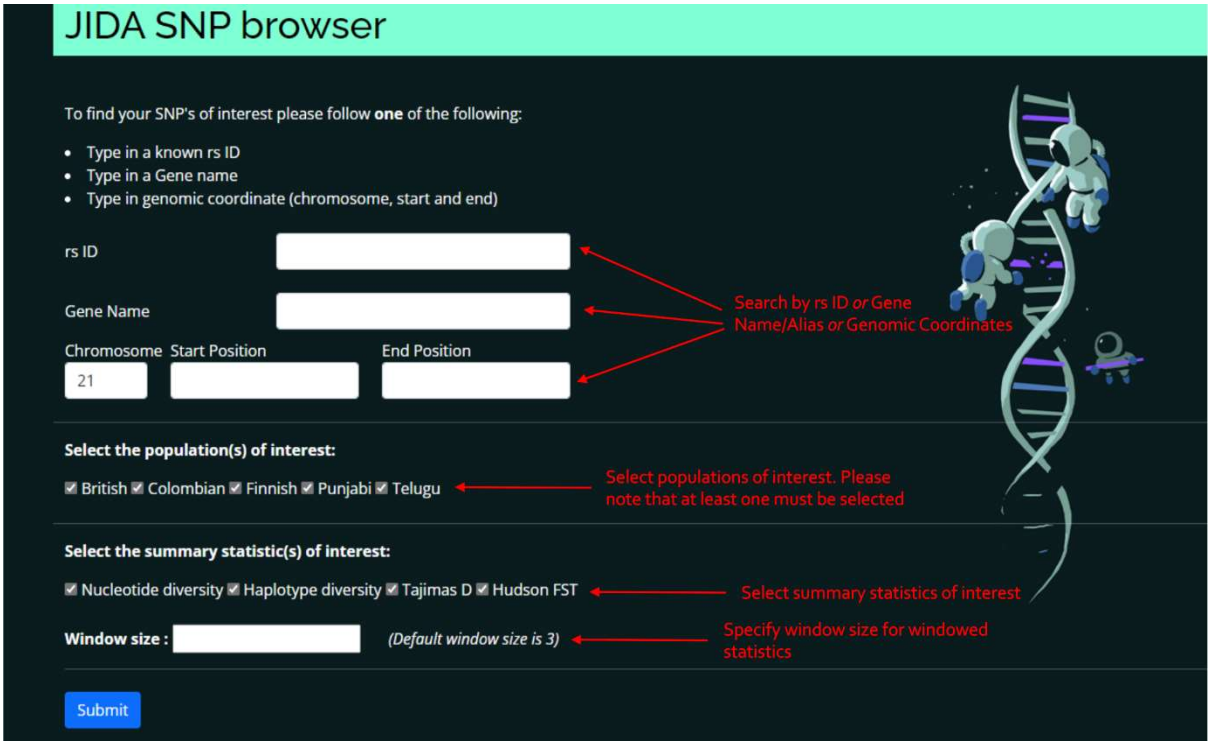


Figure 2: ***Annotated screenshot of the JIDA SNP browser.*** *The annotations describe the possible search inputs.*

Figure 2 shows the SNP browser page of the JIDA web application. Here users can input an RS id, a gene name/ alias, or genomic positions. Following this they can select their desired populations and statistics and specify a window size for windowed statistic functions. Upon clicking submit (bottom left), and after completing the necessary calculations, the web application will direct the user to an output page. The output page contains 4 main components, the SNP information table, the statistics tables, the windowed statistics tables and the download button.

2.1.1. SNP information table

Every input will produce an SNP information table (see figure 3).

SNP information :

Chromosome	Genomic Position	rs value	Population	Reference Allele (0)	Alternate Allele (1)	Reference Allele Frequency	Alternate Allele Frequency	0|0 Genotype Frequency	0|1 Genotype Frequency	1|0 Genotype Frequency	1|1 Genotype Frequency
21	19620696	rs533219480	British	A	C	0.995	0.005	0.989	0.011	0.0	0.0
21	19620730	rs377153625	British	C	T	1.0	0.0	1.0	0.0	0.0	0.0
21	19620731	rs569912440	British	G	A	1.0	0.0	1.0	0.0	0.0	0.0
21	19620817	rs73318232	British	G	C	1.0	0.0	1.0	0.0	0.0	0.0
21	19620832	rs549038365	British	T	G	1.0	0.0	1.0	0.0	0.0	0.0
21	19620890	rs567553467	British	G	A	1.0	0.0	1.0	0.0	0.0	0.0
21	19620903	rs13046215	British	A	T	1.0	0.0	1.0	0.0	0.0	0.0

*Figure 3: **Example SNP information table.** The columns include the allele and phased genotype frequencies*

2.1.2. Statistics tables and graphs

If statistics were selected and the input spans more than 1 base pair, statistic tables are produced which quote statistic values for the whole input, per population (see figure 4).

Nucleotide diversity

Show as Graph

British	0.0001572172284347842
Colombian	0.000459687664613432
Finnish	0.00015607080780788714
Punjabi	5.035254754662194e-05
Telugu	2.388287836450049e-05

*Figure 4: **Example Nucleotide Diversity table.** Other statistics tables that can be output if selected by the user include: Haplotype diversity, Tajima's D and FST.*

Clicking on Show as Graph will open another tab in which an interactive graph is presented (see figure 5). This can be downloaded as a PNG.

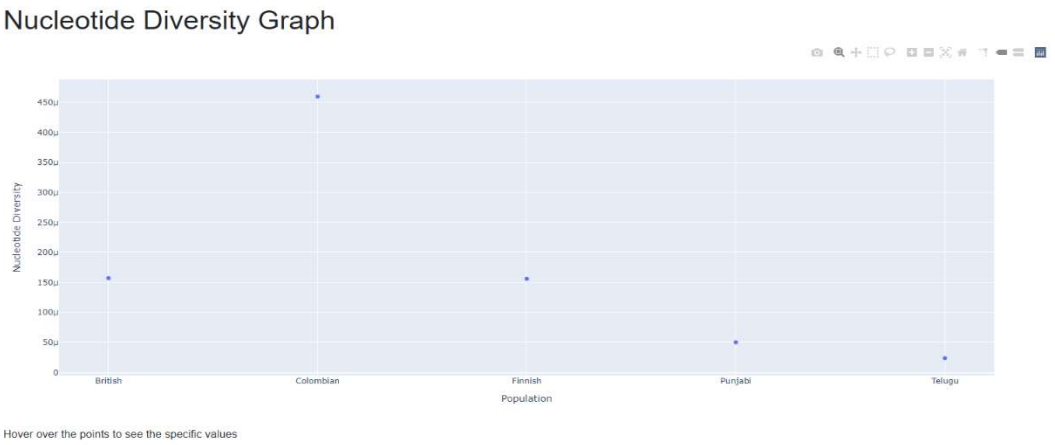


Figure 5: ***Example Nucleotide Diversity graph.*** *Other statistics tables that can be output graphically if selected by the user include: Haplotype diversity, Tajima's D and FST.*

2.1.3. Windowed statistics tables and graphs

If statistics were selected, the input spans more than 3 base pairs, and the input is greater in length than the specified window size, windowed statistic tables are produced which quote statistic values per window per population (see figure 6).

Nucleotide Diversity Window

Show as Graph

British	**[0.00011100011100011101, 0.0, 0.00022077370143668487, 0.0007512438451664971, 0.0, 0.0, 0.0, 0.0]**
Colombian	**[0.0, 0.0005257949847778042, 0.00021376582987687779, 0.0005326906567093163, 0.0003189248268324386, 0.0009487295299105518, 0.0, 0.0004275316597537557]**
Finnish	**[0.0, 0.0, 0.0001020304050607081, 0.0008892700786255624, 0.0, 0.0, 0.0, 0.0]**
Punjabi	**[0.0, 0.0, 0.0, 0.00031235126130414086, 0.0, 0.0, 0.0, 0.0]**
Telugu	**[0.0, 0.0, 9.902951079421667e-05, 0.0, 0.0, 9.902951079421667e-05, 0.0, 0.0]**

Figure 6: ***Example Windowed Nucleotide Diversity table.*** *Other windowed statistics tables that can be output if selected by the user include: Haplotype diversity, Tajima's D and FST.*

Clicking on “Show as Graph” will open another tab in which an interactive graph is presented (see figure 7). This can be downloaded as a PNG.

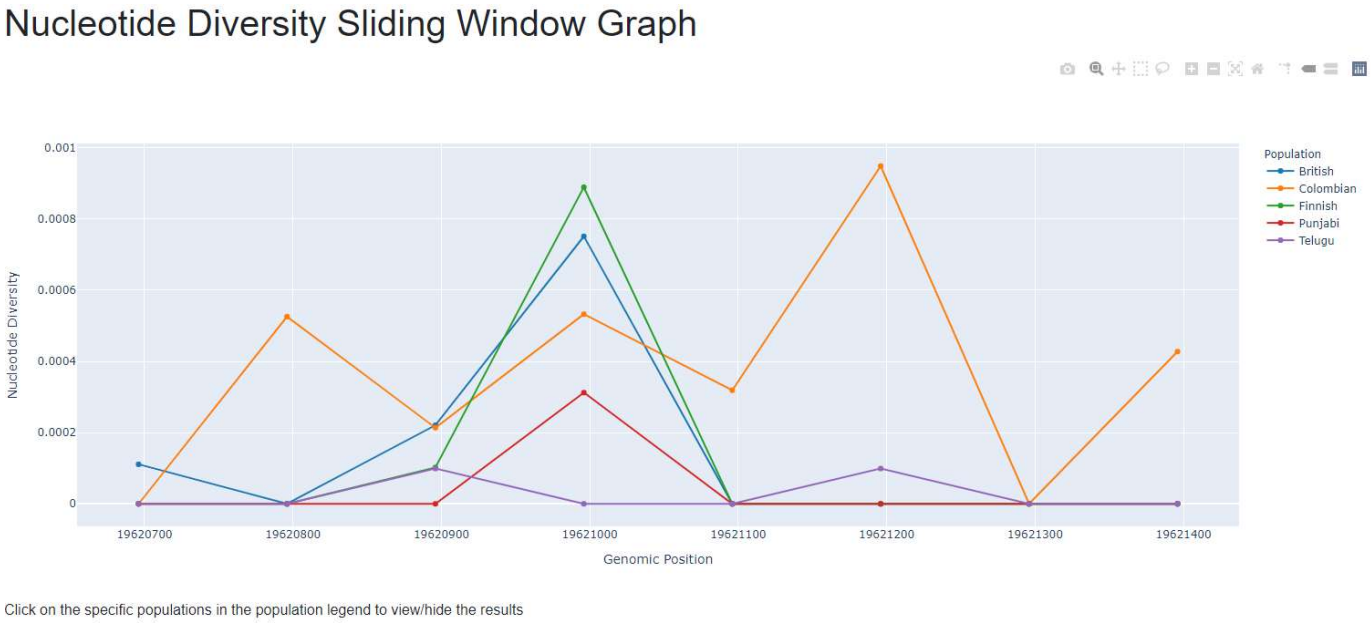


Figure 7: ***Example Windowed Nucleotide Diversity graph.*** *Other windowed statistics tables that can be output graphically if selected by the user include: Haplotype diversity, Tajima’s D and FST.*

2.1.4. Download button

If any statistics tables are produced, a Download button will appear (see figure 8, top left). This allows the user to download the statistical information in a CSV format.

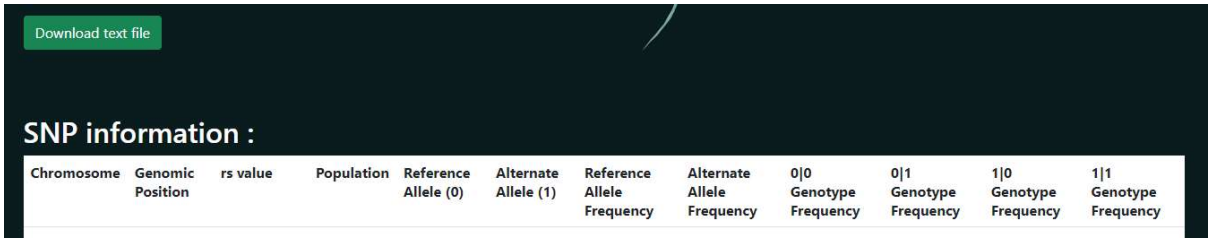


Figure 8: ***Screenshot of an example JIDA output page.*** *If statistics have been calculated, the download button will be found just above the SNP information table.*

2.2. Website architecture

The overall structure of the website, including its inputs and outputs, is represented by figure 9 below.

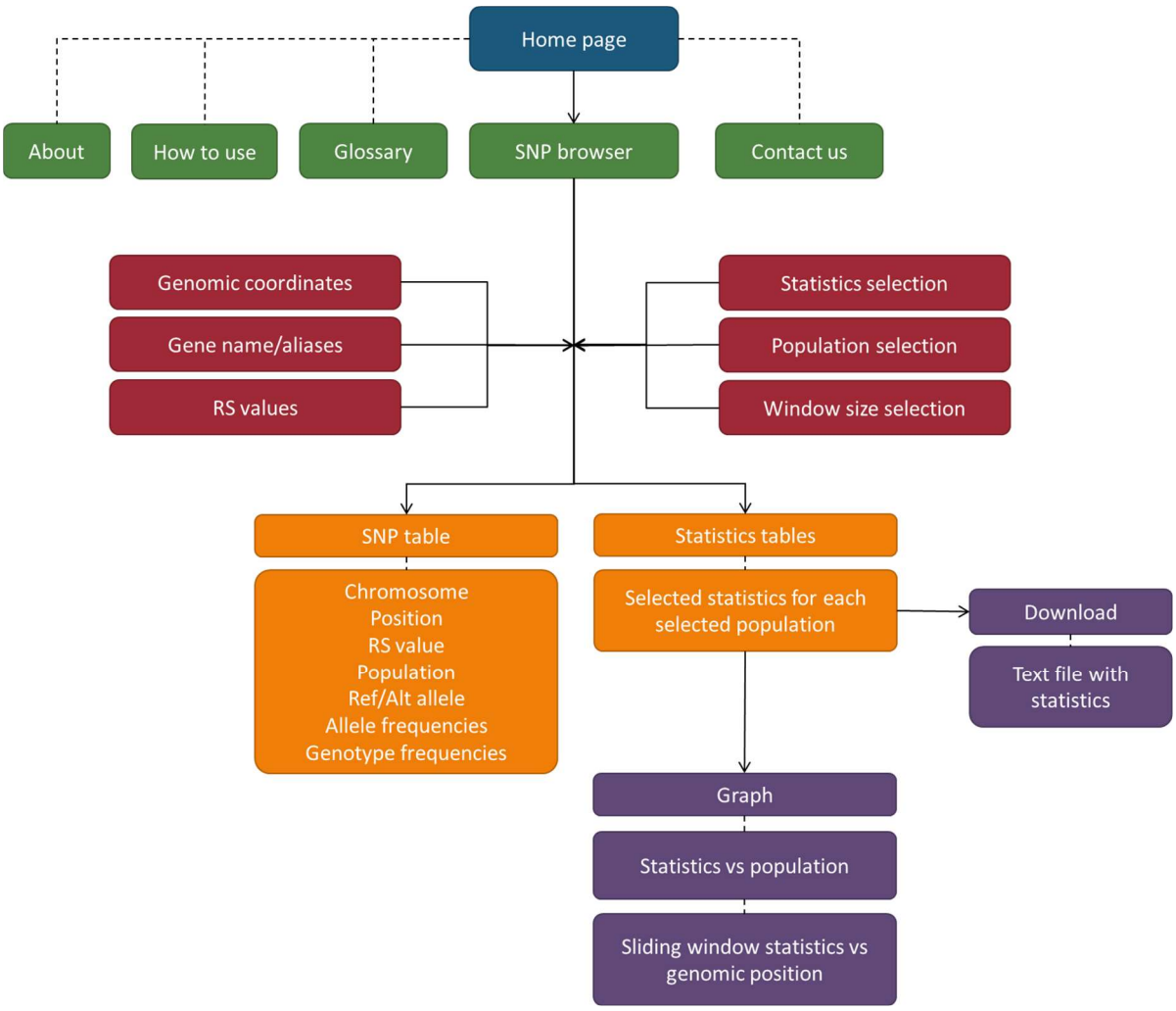


Figure 9: ***Schematic displaying the routes, inputs, and outputs in JIDA.*** *The diagram describes the routes that the website takes as it progresses with user input. The home/welcome page links all the sections of the website with a link to the SNP browser. With the user input, tables will be produced with the option to view the statistics as a graph and/or download the statistics as a text file.*

2.3. Software architecture

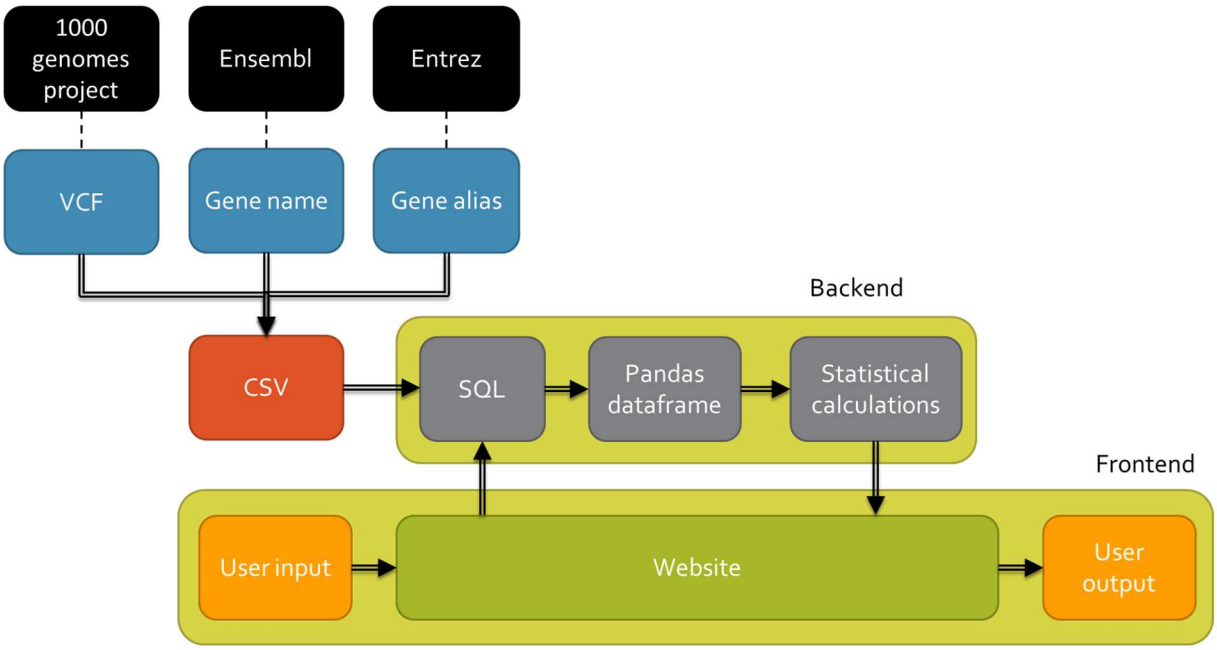


Figure 10: ***Schematic displaying the software architecture of JIDA.*** *The diagram describes the data sources used and what was extracted from them. The data was then converted into a CSV file which was fed into SQL to produce a database. Depending on the user's input into the website the panda's dataframes are produced and statistical calculations are carried out. The results are then displayed on to the website for the user to view.*

Data were retrieved from several sources to get information such as VCF files, gene names, and gene aliases. The data collected was converted into a CSV and fed into a SQL database. Depending on the queries of the user, a pandas (McKinney, 2010) dataframe is produced. The selected statistics would be calculated by functions and visually displayed on the website. The application was developed using Flask (Grinberg, 2018). The visual interface was created using a combination of Bootstrap and Jinja2. The dataframe containing the requested data was used to calculate relevant statistics and plotting appropriate graphs for each one. The user is presented with tables containing the relevant data, buttons to access interactive graphs, and the possibility to download the data in CSV format.

2.4. Installation requirements

Before running the web application locally, it is necessary to install the below python packages/modules using command line.

```
pip install Flask # version = 1.12
pip install flask-wtf # version = 1.0.0
pip install biopython # version = 1.79
pip install scikit-allel[full] # version = 1.3.5
pip install plotly # version = 5.6.0
pip install Pandas # version = 1.3.5
pip install numpy # version = 1.20.3
```

Should there be any issues installing these packages/modules, we recommend making sure your python interpreter is using python version 3.8.10. To check the python version being used, you can use the following in a command line prompt.

```
python --version
```

2.5. How to run JIDA in command line

Clone the git repository or OneDrive folder to your local machine. Within your terminal, navigate inside the repository, such that running the command *ls* (in Linux) or *dir* (in Windows) produces this output. PLEASE NOTE: if cloning the git repository, you will also have to separately download the SNP database 'SNPB (1).db' from OneDrive.

```
PS C:\Users\isabe\PycharmProjects\JIDA> ls


    Directory: C:\Users\isabe\PycharmProjects\JIDA


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        01/03/2022     13:35                static
d-----        01/03/2022     14:04                templates
-a----        02/03/2022     12:25          38388 APP.py
-a----        28/02/2022     18:14           1610 forms.py
-a----        28/02/2022     14:01           2665 GF_AF_functions.py
-a----        01/03/2022     13:35            198 Installation Requirements.txt
-a----        01/03/2022     13:49      389398528 SNPB (1).db
-a----        28/02/2022     19:20          23184 stats.py
-a----        01/03/2022     18:11          12701 Validate_Search_Functions.py
```

Once all prerequisites have been confirmed present, you can run the web application locally using the following terminal command.

```
python APP.py
```

The terminal will display a local HTTP address, please copy and paste this into your browser of preference.

2.6. Flask/Bootstrap/CSS/HTML

Flask was used to develop the web application software along with python which has the advantages of being highly flexible and compatible with latest technologies. It has high scalability for simple web applications and smaller size of the code base.

HTML was used with the CSS language to make the user interface more presentable and professional. Bootstrap (Bootstrap, 2015) was also utilised to provide a CSS and HTML design template (https://bootstrapmade.com/demo/Bethany/) for navigation bar and transitions. We used the base.html file as our home template. The background image was added as a body style (https://wallpaperaccess.com/download/dna-4k-3788423) to all the templates from the website to enhance the look of the application.

Header section contains navigation bar that is displayed on the top of this website. The name of the website 'JIDA', which is displayed as a logo followed by the links to home, about, how to use, glossary and contact us sections. Upon clicking on these links, the website scrolls to those sections.

First section – 'Home' contains the welcome information and description of what the application does. Second section – 'About' provides options the application provides for the user to select and the output format. Third section – 'Glossary' contains information on terms used in the web application such as genotype frequency, allele frequency and the summary statistics that are calculated. The user will also be able to click on each hyperlink provided as headings which redirects the user to a Wikipedia

page with more information about the relevant topic. Fourth section – 'Contact Us' contains contact details and a hyperlink to the GitHub repository.

The navigation bar contains a button on the right ('SNP Browser') that redirects the user to a new page – JIDA SNP browser. The SNP browser page contains the input fields (input boxes, checkboxes, and submit field) which are to be used to search for the desired information.

The output page contains tables (SNP Information and Statistics) that are dynamically filled according to the user's input. The graph buttons are also displayed for each statistics table similarly which directs the user to another html page that produces graphs using Plotly (6.2)

3. Data sources and Processing

3.1. Selecting VCF data and converting to population specific VCFs

For our database, we used a VCF of chromosome 21 that was generated as part of the 1000 genomes project (Auton et al., 2015). We downloaded the raw file from the NCBI FTP site (ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/). Beforehand, VCFs were downloaded from the Ensembl FTP site (http://ftp.ensembl.org/pub/), however, these were set aside once it became clear they did not contain phased genotype data (Clarke et al., 2012)).

The file selected (https://ftp-trace.ncbi.nih.gov/1000genomes/ftp/release/20130502/ALL.chr21.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.VCF.gz) was chosen because it contained the data needed for our analysis (Auton et al., 2015). Most SNPs were named, the genotypes were phased (which is necessary for the downstream calculation of nucleotide diversity, Tajima's D, and FST) and contained sample sufficient sample counts for the selected populations (n ≈ 100).

From this file, BCFtools was used to extract population-specific data (Danecek et al., 2021).The file was first filtered to obtain only SNP data using the *bcftools view –v snps* command. Next, population-specific VCFs were produced by using the command *bcftools view –S. --force-samples* which extracted the information of specific samples only. This was done by passing text file containing the sample identifiers from a specific population to the function. BCFtools was used because its versatility to manipulate VCF files and the very extensive documentation available to the users.

3.2. Converting population VCFs to CSVs

3.2.1. VCFs to pandas dataframes

`VCFtoPandas(infile)`

Within python VCF files could be opened and converted to pandas dataframes using the custom-made function VCFtoPandas(). This was an especially important step for reading the contents of the VCF file, and allowed further data processing that could be facilitated using pandas' (McKinney, 2010) vast function set.

3.2.2. VCFs to SNP table CSVs

`VCF_to_SNP(inVCF, outCSV, start, end)`

The creation of the SNP table CSV was completed within python using the custom-made function VCF_to_SNP() which itself used VCFtoPandas() for its first step. This function took in the path to a single population VCF file name (inVCF), a path to an output CSV (outCSV), and start and end indices. The function returned a CSV which contained the chromosome number, genomic position, RS id,

alternate and reference alleles for each SNP present in the original VCF, between the start and end indices.

The reason these CSVs were generated for only a subset of the VCF data at a time is due to how long it took for the function to run. Since providing the function only a subset of the VCF data at a time meant it output faster, any issues with the output could be more rapidly resolved, and there was less risk of a function being unexpectedly halted partway through execution.

In total, 6 SNP table CSVs were generated in this way, together spanning the whole of the chromosome 21 VCF. Only the Punjabi population VCF was used to generate these CSVs. This is due to, as all the population VCFs were derived from the same original VCF, the variants in each population VCF are the same.

3.2.3. SNP information CSVs

`VCF_to_SNP_characteristics(inputlist, outputlist, popcodelist, start, end)`

The creation of the SNP characteristics table CSV was completed within python using the custom-made function VCF_to_SNP_characteristics() function, which itself used VCFtoPandas() for its first step.

This function took a list of paths to population VCF files, a list of paths for output population CSVs, a list of three letter population codes and start and end indices.

The function returned a CSV for each population which contained the RS id, the sample count and the phased genotype counts per variant. Finally, combining all these CSVs created the final output, which contained variant information per population, per variant. In total, 4 SNP characteristics table CSVs were generated in this way, together spanning the whole of the chromosome 21 population VCFs.

Creating a CSV with sample and phased genotype counts per variant and per population was necessary, as this information were needed for the calculation of statistical values later on.

3.3. Gene name data

`genomic_position_to_gene_name(chromosome, genome_position)`

Ensembl (Howe et al., 2020) is a genome browser that contains annotated genes and is a well-developed database. It contains data for several species and is especially developed for vertebrates. Thus, Ensembl was utilised to get the gene names for each SNP in the VCF. To integrate Ensembl in python PyEnsembl (Pypa, 2015) was used which allows rapid querying between the two interfaces. The RS values in the VCF are not unique as they could be repeated in the genome, however the position number for each SNP would be unique. Therefore, by using PyEnsembl, the chromosome number and genomic position of interest were inputted and the function produced a dictionary which contained the gene name. This function was utilised to produce a row in the CSVs containing the gene names for each SNP.

3.4. Gene alias data

`gene_alias(gene_name)`

Entrez (Maglott et al., 2010) is a molecular biology database which covers over 20 databases allowing integration of nucleotide and protein sequence data. An essential property of Entrez is the links which combines many databases and thus this is vital in producing a range of gene aliases from a single gene name. A gene alias function was developed from a function posted in BioStar (Parnell et al., 2011) and

was altered to allow the production of a CSV file of the gene aliases. The function used BioPython (Cock et al., 2009) and searched the Entrez database for the gene name entered in the function and it returned a list of the gene aliases. A CSV file was produced with the gene name in a column and adjacent was the gene aliases found by the Entrez database.

3.5. CSVs to SQLite3 database

CSVs were minorly adjusted to fit the table titles from the database schema. This involved merging or appending columns within python, or directly in Microsoft Excel where size allowed. Specific editions to CSVs included: changing column titles with the PD_change_column_name.py script, writing a CSV with the unique values of gene names using Gene_table.py, and finally, writing a CSV where all positions where SNPs were found were appended next to their corresponding chromosome and (if present) a gene name with gen_coor table script.py. The population and chromosome CSVs were produced using Microsoft Excel and filled using the information directly from the 1000 Genomes Project. Using Database_script.py, the final versions of the CSVs were imported as pandas elements and passed into the corresponding tables in the database.

4. The Database

4.1. The Schema

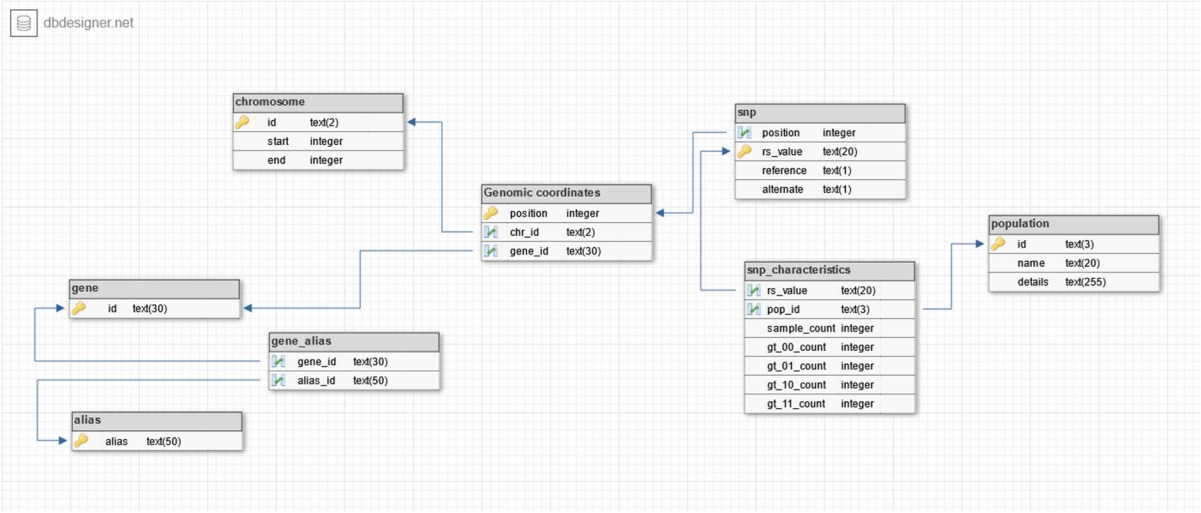


*Figure 11: **SNPB database schema.** Graphic representation of the tables and relationships for the data stored within the database. The database can be separated into three sections: genomic coordinates, SNP data, and gene data. Keys refer to primary keys, green links refer to foreign keys. Blue arrows show the relationships between primary and foreign keys.*

The database schema consists of 8 tables that encompass the necessary data to link from genomic positions/coordinates to SNP characteristics and their population (Figure 11).

4.1.1. Schema Tables

Genomic coordinates: The central table from the schema as it connects most of the tables together. It contains three columns:

- *Position*: the genomic coordinate in a given chromosome where a SNP has been registered in the original VCF file. – Primary key
- *Chr_id*: the chromosome to which the genomic coordinate belongs to – Foreign key
- *Gene_id*: name of a gene in case it belongs to that genomic coordinate – Foreign key

Chromosome: table containing chromosome information:

- *id*: contains the unique chromosome number – Primary key

- *Start*: the genomic start position
- *End*: the genomic end position

This table is linked to the main genomic coordinates table via the chromosome number via a one-to-many relationship.

SNP: The SNP table contains the general information of a SNP:

- *Position:* genomic position in which the SNP was reported – Foreign key
- *rs_value*: rs name or id contains the rs identifier of the SNP – Primary key
- *reference*: one letter nucleotide symbol for the reference allele
- *alternate*: one letter nucleotide symbol for the alternate allele

This table is linked to the main **genomic coordinates** table via the position number via many-to-one as the positions may contain more than one SNP.

SNP characteristics: This table contains the population specific information of a SNP, including genotype counts.

- *rs_value*: rs name or id contains the rs identifier of a SNP – Foreign key
- *pop_id*: contains a 3-letter identifier of the specific population the information of that SNP belongs to – Foreign key
- *sample_count*: the number of samples that were taken for the given population
- *gt_00_count*: contains the count of samples with the 0|0 genotype.
- *gt_01_count*: contains the count of samples with the 0|1 genotype.
- gt_10_count: contains the count of samples with the 1|0 genotype.
- *gt_11_count*: contains the count of samples with the 1|1 genotype.

This table is linked to the **SNP** table via the rs value in a one-to-many relationship. This means that for every SNP, there are multiple populations with distinct characteristics.

Population: This table contains information about the population:

- *id*: Three letter code for a specific population (GBR, FIN, PUN, TEL, COL) – Primary key
- *name*: full name of the selected population
- *details*: short description taken from the 1000 genomes project population description as to what the population is where it is based.

This table is connected to the **SNP** via the id column in a one-to-many relationship as a population can be present in many SNPs.

Gene: contains the gene names of the genomic coordinates

- *id*: gene name based on ENTREZ – Primary key

This table is linked to the **genomic coordinates** table via the gene id. This is a one-to-many relationship as one gene can be present in many positions/coordinates.

Alias: contains the alias for the gene names in the selected genomic coordinates

- *id*: alias names based on ENTREZ – Primary key

These two tables share a many-to-many relationship meaning that a gene name can have multiple aliases but also that a single alias can belong to multiple genes. For this the **gene_alias** table was created to bridge this relationship

Gene_alias: contains the gene names that correspond to one or more gene aliases registered in ENTREZ:

- *gene_id*: gene name based on ENTREZ
- *alias_id*: alias associated with the gene

The database could have been designed with less tables and entries for the current dataset. However, it was designed to be scalable and store more chromosome data in the future with minimal changes to the structure and relationships.

4.2. SQLite

SQLite is an open source embedded relational database management system. SQLite was chosen as it allows to have a portable database that could be shared with multiple developers as a file instead of a client-server database engine like PostgreSQL (Junyan et al. 2009). The estimated size of the whole dataset for only one chromosome was small (< 500 Mb), thus the need for a larger database was not a priority. SQLite also provided a very developer-friendly interface which allowed to create and connect to the database with Flask with ease. In addition, it does not require further installation for users who have python 3 or higher. Query syntax is simple compared to other database managers making it easy to track and make any changes as the database evolves. SQLite is suitable for low-medium flow applications which is what we have aimed for with this app (Devanthon-Team, 2021).

4.2.1. Populating the tables

The database was created, populated, and tested using the Database_script.py file. This file is crucial as it contains the tables, indexes, and queries the database is composed of. The 3 components were kept in the same script to facilitate debugging and monitoring changes. The script was used locally to monitor query speed performance and correct data delivery independent of other back-end processes.

4.3. Queries

To retrieve information, 4 different queries were written to create a view with the same columns. The complete set of queries can be found in the Validate Search Functions.py script

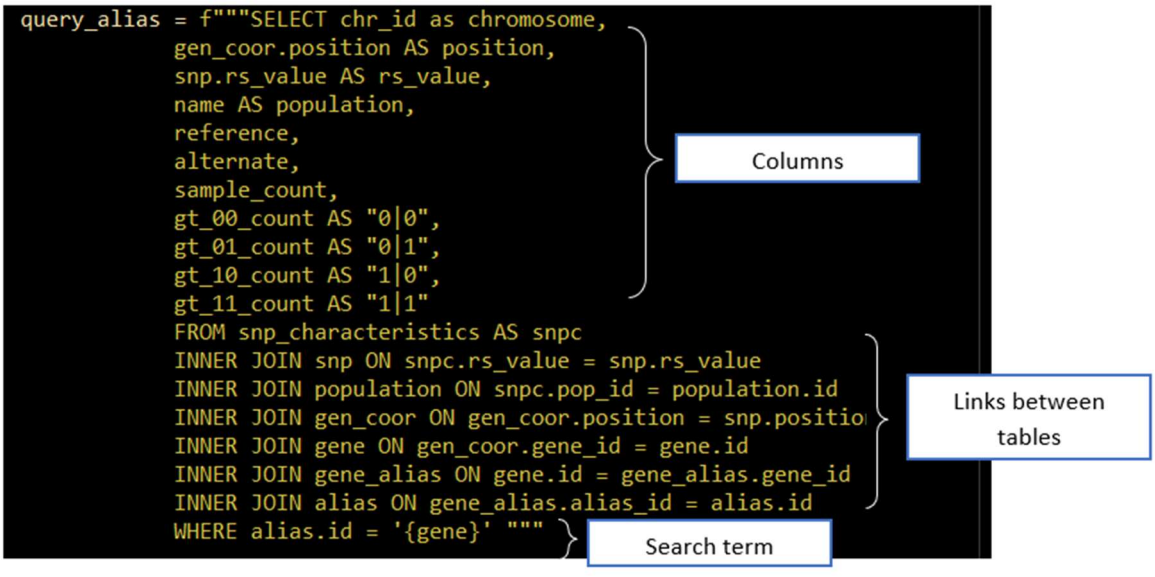


*Figure 12: **Query example**. The syntax for the queries in the application. Query contains a SELECT, FROM, and WHERE sections where columns, tables and search terms are selected, respectively. In the application, 4 queries were written in very similar fashion to find gene names, gene aliases, rs values, and positions.*

The query is divided in three segments the **SELECT**, **FROM,** and **WHERE**. The **SELECT** statement returns the column needed for display or for calculations. The column was specified as either "column_name" or "table.column_name" (Figure 12). The latter was used if the name was present in more than one stable. The AS statement was used to select the column name in the output as shown in Table 1.

*Table 1: **SQL query column names and output column names.** The SQL column describes how columns are referenced in the query. The "To Flask" column shows the column names as delivered to the application (user).*

Column title	
SQL	**To Flask**
Chr_id	Chromosome
Gen_coor.position	Position
SNP.rs_value	Rs value
name	population
Sample_count	Sample_count
Gt_00_count	0|0
Gt_01_count	0|1
Gt_10_count	1|0
Gt_11_count	1|1

The **FROM** statement selects the table(s) the data is drawn from. To call the related records the **INNER JOIN** and **ON** statements are used. The first **INNER JOIN** takes the table referenced after **FROM** and joins it with the table that follows. The **ON** statement is used to specify where the relationships between tables are in the format "*table1.column = table2.column*". In the example above, *SNP* is joined to *SNP_characteristics* (also named as *SNPc*) as they both share an *rs_value* column (Figure 12). The following **INNER JOIN** statements identify any table that was previously referenced.

Inner joins were selected as this is the only type of join that returns all records that have matching values across all columns. This is important since some positions/rs values are not associated with a gene as well as some genes do not have an alias.

Finally, the **WHERE** statement is where the search term is introduced. In this specific example, the **WHERE** was set to a specific gene/alias input (Figure 12). In the position query, the **WHERE** statement was combined with **BETWEEN** and **AND** to search for positions between a certain range.

4.3.1 Indexing

SQL indexes were used to substantially increase query execution and data retrieval speed. In our database, we used single and compound indexes to accelerate our queries. Indexes were placed strategically in the columns where the searched values were stored. The full structure of indexes can be found in the Database_script.py section 3. The selected **tables**(*columns*) were:

-**Gen_coor**(*position*), **gene**(*id*), **alias**(*id*), **gene_alias**(*gene_id*, *alias_id*), **SNP_characteristics**(*rs_value, pop_id*). These indexes were carefully selected as the database size increases for each index created and make inserting and deleting data more time consuming for the database. The data input into the database is not expected to be recurrent, therefore we anticipate the indexes will not represent a problem in this regard.

5. Statistical Analysis

5.1. Frequency calculations

5.1.1. Allele frequency

The allele frequency function calculates the allele frequency based on the samples that exhibit a specific genotype (GT). The function receives the count of each GT, adds the corresponding allele values together, and divides it by the total sample count (which is the sum of all GT counts). The function assumes diploid samples. Thus, it only receives 4 inputs in the following order: GT 0|0, 0|1, 1|0, 1|1. The function returns a dictionary with the allele number as key and the frequency as value. This function was written as the allele frequency per SNP was not calculated properly when the population specific VCF files were produced. This function was written by the developers because it required simple calculations and the input and output could be tailored to the needs of the app, instead of adapting the app to external functions. In addition, this was chosen over the BCFtools allele frequency calculation as we used the genotype counts for our calculations while their function used sample count and allele count only, which would give a less precise measure.

5.1.2. Genotype frequency

This function calculates the genotype frequency based on the samples that exhibit a specific genotype (GT). The function receives the count of each GT and divides it by the total sample count (which is the sum of all GT counts). The function assumes diploid samples. Thus, it only receives 4 inputs in the following order: GT 0|0, 0|1, 1|0, 1|1. The function returns a dictionary with the allele number as key and the frequency as value. Like the allele frequency function, this function was calculated by the developers to tailor the needs of the app and it was not part of the BCFtools calculations package.

5.2. Statistics for multi-base regions

Scikit-allel (Miles et al., 2021) is a package which is utilised for analysis of genetic variation data and contains several statistical functions. We choose the Scikit allel package as it was integrated in python and thus allowed us to complete the statistical calculation without the need of R. The statistical calculations required genotype and haplotype arrays which could be easily produced from the VCF file using the SAMPLES column. Using the statistical functions, the sliding windows were also calculated and the results were plotted graphically. All the statistical functions utilised were also carried out in published articles which provided reassurance that they were producing reliable results. The results produced from all the statistical tests were as expected and in a range of values that we expected.

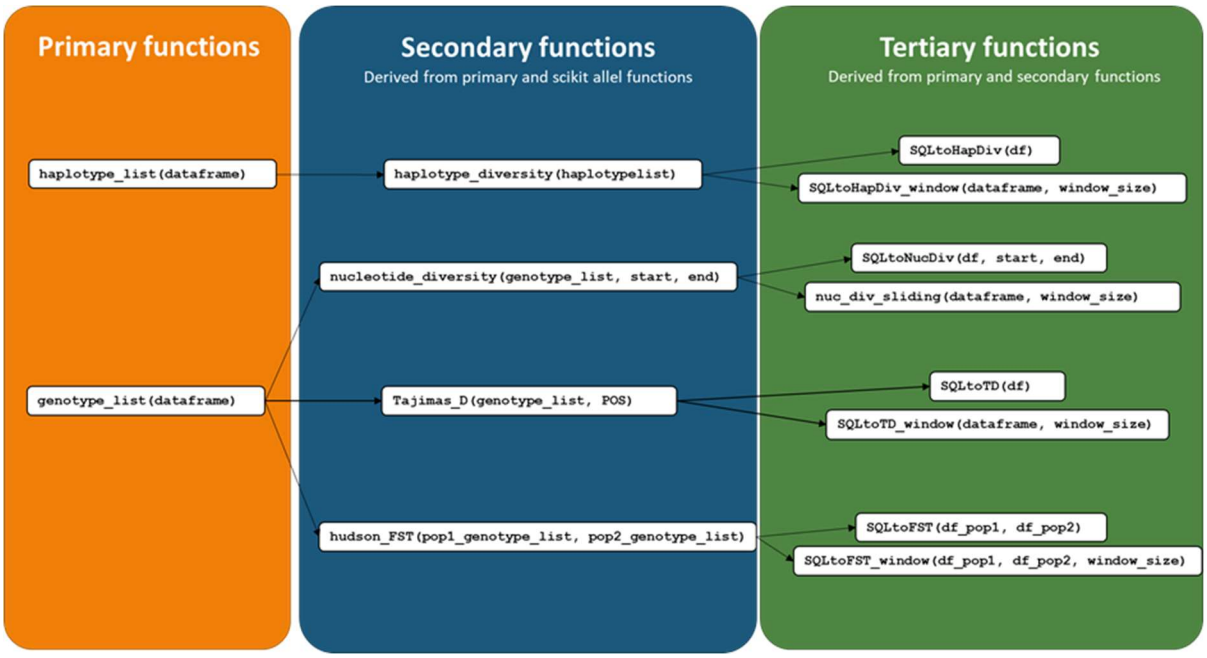


*Figure 13: **A summary of the multi-base region statistics.** Primary functions (orange) produce the genotype and haplotype lists. The secondary functions (blue) calculate the statistics. The tertiary functions (green) combine the primary and secondary functions to produce a single output.*

5.2.1. Constructing the genotype and haplotype list

For the statistical calculations the input required a genotype or haplotype list which would be later converted into an array. The information was calculated using the sample count and phased genotype counts stored within the SQL database.

`genotype_list(dataframe)`

`haplotype_list(dataframe)`

For the statistical calculations the input required either a genotype or haplotype list. Once a user's query had been passed through it returned a pandas dataframe which contained columns with counters for the phased genotypes. For each row the counter was extracted, and the relevant phased genotype was appended to the list depending on the counter value. For the haplotype list the counter values were again extracted, but for each phased genotype both alleles were appended to the haplotype list. The genotype and haplotype lists produced were returned by this function and utilised by the statistical functions below.

5.2.2. Nucleotide Diversity

`nucleotide_diversity(genotype_list, start, end)`

Nucleotide diversity is a measure of genetic diversity and can highlight the genomic regions of functional importance (Tatarinova et al., 2016). To calculate the nucleotide diversity the genotype list, start and end positions are required. This function calculated the nucleotide diversity within the region selected by the user. The genotype list was converted into an array and a list of the variant positions was produced. Both are inserted into the sequence diversity function from allel (Miles et al., 2021) to return the nucleotide diversity. The function calculated the average proportion of sites which differed between randomly selected pairs of chromosomes. An article showed great alignment in the statistical results between allel and different packages such as PopGenome so it is a reliable function to use (Korunes and Samuk, 2021).

SQLtoNucDiv(df, start, end)

The genotype list and nucleotide diversity function were collected in another function where a dataframe is inserted and converted into a genotype list. Then the genotype list, start and end values were inserted into the nucleotide diversity function and the value was returned.

5.2.3. Haplotype diversity

haplotype_diversity(haplotype_list)

Haplotype diversity is the probability that two randomly sampled alleles are different (de Jong et al., 2011) and is affected by several processes such as mutation and recombination (Stumpf, 2004). Values close to 1 shows the user that the alleles are different whereas values close to 0 display similarity in alleles. This function calculated the haplotype diversity (H) by using a haplotype array. From the haplotype list function, the haplotype list was produced and converted into a haplotype array. The haplotype array was inserted into the haplotype diversity function by allel (Miles et al., 2021) which produced the required statistics. The function retrieved the number of haplotypes and the haplotype frequencies before inserting the values into the haplotype diversity formula. The statistic values produced by the function are within the range of 0 and 1.

SQLtoHapDiv(df)

The haplotype list and haplotype diversity functions were collected in another function where the dataframe inserted produced a haplotype list. The haplotype list was placed into the statistics function and it returned a single haplotype diversity value.

5.2.4. Tajima's D

Tajimas_D(genotype_list, POS)

Tajima's D is a neutrality test used to distinguish if a population is evolving neutrally or under a non-random process.

A value of zero indicates there is little evidence of selection of certain alleles over others in the population, and that population is evolving neutrally. Negative values would indicate to the user that the population is not evolving neutrally, but rather is undergoing purifying selection wherein deleterious variants are being removed from the population. Positive values indicate that balancing selection is occurring in the population, wherein multiple alleles are actively maintained in the gene pool.

This function calculated Tajima's D value over a given region. The input is a genotype list, produced from the genotype list function, and a list of the variant positions. The genotype list is converted into a genotype array and the allele counts are calculated. Into the Tajima's D function by allel (Miles et al., 2021) the allele count and the list of variant positions was inserted. The minimum number of segregating sites was set to 1 as Tajima's D would be calculated for regions where there is minimal segregation. Then by inputting the values through the function the value was returned. The function produced by allel has be utilised by published articles (Ralph et al., 2020) and thus provides reassurance that the function produces reliable results.

SQLtoTD(df)

The genotype list and Tajima's D function were collected in another function. Here the dataframe was converted into a genotype list and then the function calculated the variant positions required for the statistics function. After calculating Tajima's D if any values were returning as NaN the function returned a value of 0.

5.2.5. Hudson FST

Hudson_FST(pop1_genotype_list, pop2_genotype_list)

The fixation index (FST) is measure of population differentiation and is used to evaluate how much overlap there is between allele frequencies in two populations. Values close to 1 show no overlap in allele frequencies and values close to 0 show strong similarity between the two populations. So, this statistic is useful for the user to evaluate population similarities.

The function was used to calculate the Hudson FST and requires the genotype lists for both populations. The genotype lists were produced from the genotype list function. Hudson FST produces an estimate which is the average of FST and they are independent of sample sizes even when FST is not identical across the populations (Bhatia et al., 2013). The numerator and denominator are both unbiased estimates and the ratio produced would be a converging value as the number of SNPs increase (Bhatia et al., 2013). An article showed that the Hudson FST values were more stable compared to Weir and Cockerham FST where the FST values were greatly diverging with identical populations and identical individuals (Bhatia et al., 2013).

Using the two genotype lists the function produces two genotype arrays for the selected populations and the allele counts are calculated for both populations. The allele count values are placed into the Hudson FST function from allel (Miles et al., 2021) where the numerator and denominator are produced. By summing the numerator and denominator values, by using numpy (Harris et al., 2020), and then dividing these values the FST statistic was calculated. If the denominator is 0 then the FST calculation would be null, therefore in these specific cases the overall FST value output in 0.

The Hudson FST calculation can also produce negative values which are usually unexpected. Rather than changing the values to 0 they remained in our calculations as negative values to show there is more variation within than between the populations and the loci is considered to have no population differentiation.

SQLtoFST(df_pop1, df_pop2)

The genotype list and Hudson FST functions were collected in another function. Here using the two dataframes the genotype lists for the two selected populations were produced. These genotype lists were inputted into the FST function and a single value was returned.

5.2.6. Windowed Statistical Functions

nuc_div_sliding(dataframe, window_size)

SQLtoHapDiv_window(dataframe,window_size)

SQLtoTD_window(dataframe, window_size)

SQLtoFST_window(df_pop1, df_pop2, window_size)

For larger genomic regions, such as a gene, it is often beneficial to measure a statistical value in windows across the region rather than calculating one value for its entirety. This windowed approach allows researchers to identify particularly high or low diversity areas of the region they've selected, which is often more biologically informative than simply having an average value.

For all our windowed statistic functions, the parameters are the dataframe (or dataframes) produced by the SQL query and the window size, measured in base pairs. We have chosen to allow the user to select their preferred window size because there is no window size that will be appropriate for all

possible genomic region sizes. Additionally, this means there is no minimum computational load for the website that the user will be forced to sit through, as they can select as large a window size as suits them.

We are measuring our windows in base pairs (Beissinger et al., 2015) for these functions as this allows us to mark the x-axis of our windowed statistic graphs with the genomic positions, which is more biologically meaningful than a count of variants. Due to this decision, we wrote the windowed statistic functions instead of using pre-written ones from scikit-allel.

6. Website searching and outputs

6.1. Searching fields

When the input is given into html page, the generated dialog boxes checkboxes and button in SNPForm are imported from the python file forms.py. Flask form is used to create a class and wtforms (wtforms, 2012) are used to specify the field and validate input. Each box and button were created under the same class as only input from one is required. The gene name and rs value are taken as a string. The genomic coordinates and window value are taken as an integer field along with submit as submit field and checkbox as Boolean field. As only chromosome 21 is used in the database the select chromosome field is given only one option.

6.1.1. Filtering and Validation

Filtering, validation, and database querying occur within the script Validate_Search_Functions.py The *search_term*() function takes the data from the forms (gene, rs, start, and end). The function checks for empty forms, simultaneous entries (for example: an attempt to search by rs value and gene name at the same time), among other invalid inputs. If the input was correct, the function returns the values the user passed, if not an error message is stored and returned as a flash warning. This was done to filter the user's input into the infrastructure of the SQL queries as well as to provide custom feedback messages to the user.

Validation and query selection occur at the *query_select*() function. This function takes the output from *search_term*() and the data from the forms. Within this function, the 4 pre-written SQL queries (4.3 Queries) are stored in a dictionary. The function checks the output from *search_term*() and the input from the forms and selects the appropriate query to execute based on input. To distinguish between alias and gene queries, there is a pre-written list of gene names where the gene form input is checked for first. Since the queries are long strings, including them directly in the application code would make readability and edition more complex and error prone. Having the queries and the validators in a different script from the application made changes to queries and error debugging more efficient for the developers. The final function *window_check*() checks the sliding window value and makes sure it is suitable for the information requested by the user. If it exceeds the positional length of the SNPs requested, it displays a message and suggests a new window size. This was done to provide useful feedback about window size tailored to each specific request.

Once the inputs are selected and submit button is clicked the app route for search function starts running the queries and collects the data from the SQL database. The data collected from the SQL queries is then converted into a dataframe using pandas.

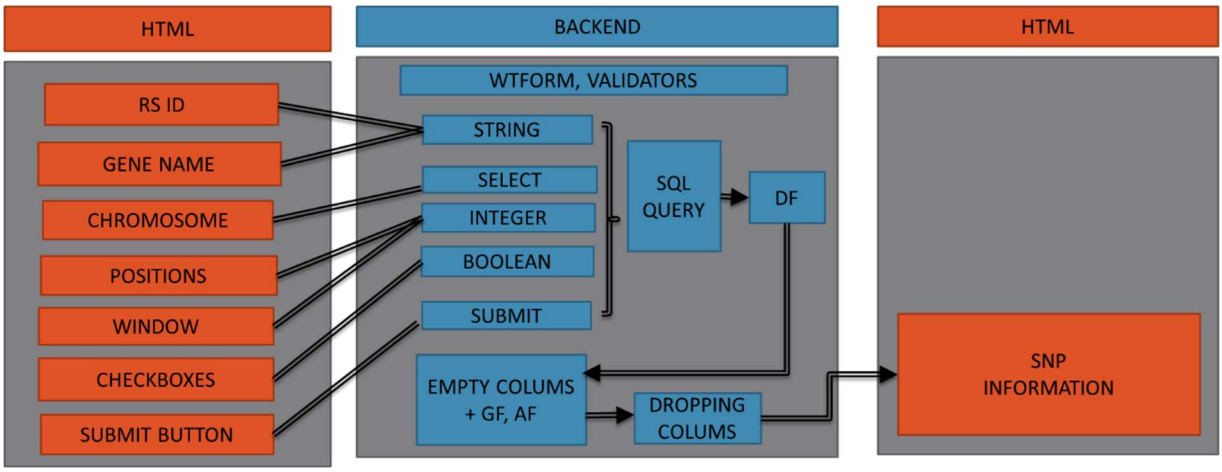


Figure 14: ***Schematic showing how user input is validated, processed data output.*** *Types of data taken as input, how they are validated, queried from SQL, converted to a dataframe, addition of genotype and allele frequencies columns and processing before returning the output.*

Empty columns are created for calculations of genotype and allele frequencies, using the functions imported from *GF_AF_functions* (*gtfreq, allefreq*). The columns names that are not to be displayed on to the output are dropped, the dataframe is then rendered on to the table created on the output page (SNP Information) using a for loop.

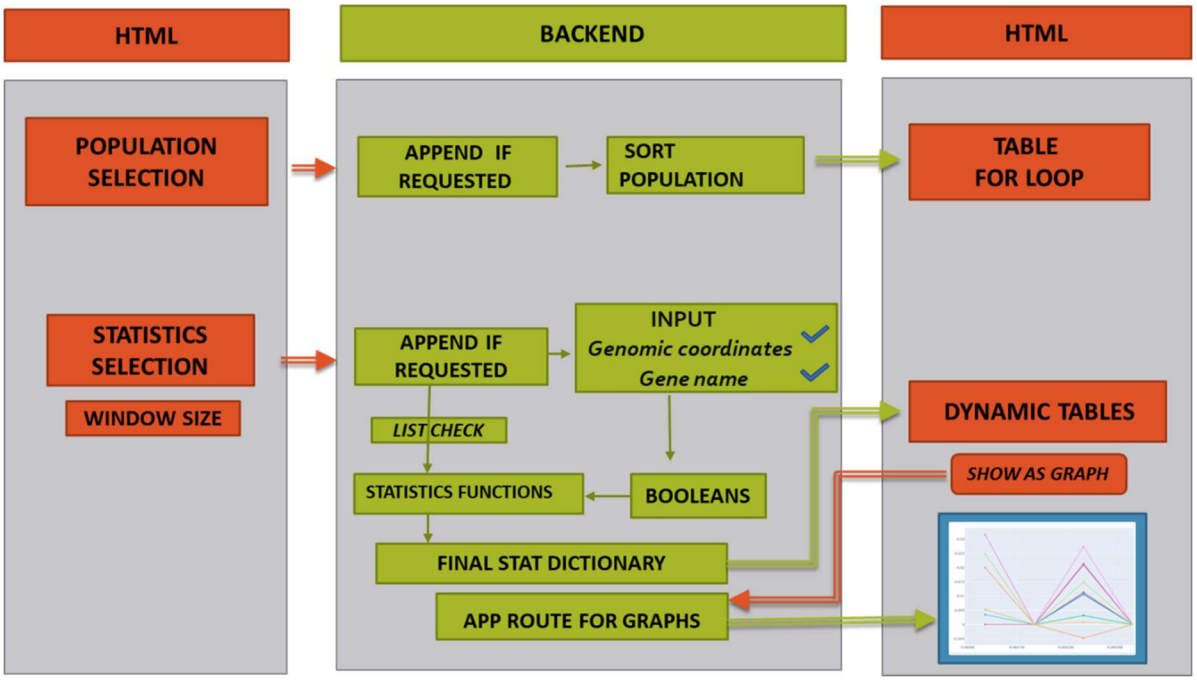


Figure 15: ***Schematic showing the route from data input to the data input - data process - data output of population and statistics input routes.*** *How the input from the Checkboxes is collected and used for the processing and statistics calculation before presenting dynamic tables on to the output page along with button that redirects to a new page and plot the graph.*

The input from the population checkboxes is taken into a list, using which the dataframe is filtered accordingly to contain only selected populations. The data is then sorted by population that is all the SNP information for one population followed by the next selected population. The dataframe is then feed into the table (SNP Information) on output html using a for loop. Similarly, the input from the statistics checkboxes is also taken in a list. Booleans are used which are later made true if the list appends the statistic. The statistics functions calculate the values, which are then taken into

6.2. Sliding window graphs – Plotly

Plotly (Plotly-Technologies-Inc., 2015) is a python library which can be utilised to produce graphs and charts. It produces interactive graphs where the user can hover over results to see the precise values. There is a toggle bar which allows the graph to be downloaded as a PNG and the user can zoom in/out of the regions of interest. The application returns two graphs: the statistical values against the populations selected by the user and the sliding window statistical values against the genomic position. The graphs and html files were produced using the guidance of the website Towards Data Science (Jones, 2021). The graph output was produced as another app route on the website, the graphs also open in another tab allowing the user to access the results tables again.

The first graph was produced using Plotly Express which contains functions required to produce common and simple figures. For each statistics a pandas dataframe was developed that was fed into the plotly express scatter function and the graph was returned in another tab. The graph describes the statistical values for each population selected, this would allow the user to compare the overall statistical results between the populations.

The second graph for sliding windows was producing using go.Scatter. This function provided ease in produce an empty graph and then adding graphs for populations specified by the user. The genomic position, x-axis, was calculated depending on the users input and the window size selected. The genomic positions axis would allow the user to hone into regions of the genome they find of particular interest and would like to further research. The population legend produced in the second graph is interactive and the user can select/deselect population graphs of interest. The axis of the graph would automatically alter to produce a clear distribution of the results. For each population specific colours were selected to allow continuity within the graphics, however during the FST sliding window graphs the colours will default to the colours by Plotly.

6.3. Text file downloading

The download button is only displayed if the output has more than one rs value. A function is defined to take the items inside global variable that has been created in the app which stores the summary statistics data, using which a new dictionary that contains names of the statistics with the names of the populations and values it produced is created. The data is converted into a pandas dataframe and then into a CSV file using buffer function. Data can be kept as bytes in an in-memory buffer when we use the io module's Byte IO operations (Python, 2007, JournalDev, 2021), which is returned as a text file with data separated with commas.

7. Limitations

7.1. Deployment

The application only runs locally, which requires the user to store the database, the repository, and manually run the application through a terminal. This can be challenging for users who have little experience using bash. Deploying the application would ensure the user can access the content through a URL address in their browser.

7.2. More gene aliases

The database only stores gene alias data for gene names stored in Entrez NCBI as explained in function (Section 3.4). Regardless of this limitation, more gene aliases can be retrieved from UniProt and HUGO Nomenclature Committee databases given the appropriate functions exists. Once that is obtained, the data can be appended to the current database and provide a more extensive range of aliases per gene name.

7.3. Faster user outputs

Currently, the queries and the data processing are fast which allowing swift statistical calculations. However, the visual interface and displaying the statistics requires more time than expected. This can be problematic for users as for larger genes or longer genomic positions the time taken for the bootstrap table to be displayed is time consuming. This could generate a poor user experience and overshadow the good quality data offered by the application.

7.4. Pagination

The results are displayed in population and statistic-specific tables that can get particularly long if the returned information is very large (for example, when searching for a gene). This requires the user to scroll for long times which could make the user experience less pleasant. This can be solved in the future implementing Django tables which enable pagination for the existing tables.

8. Future development

8.1. More chromosomes

In its current state, JIDA only displays information for chromosome 21. However, the database structure, query system, and data processing within the app are designed to incorporate data for more chromosomes without incurring major changes in the overall back-end structure. To add further chromosome data, the pertinent VCFs would need to be filtered and converted into CSVs with the existing pipeline. The schema contains existing tables compatible with these CSVs and the queries would need an additional JOIN statement to retrieve data from the corresponding chromosome (Section 4.2). In terms of the front-end, the interface has been equipped with a select field to choose a chromosome which can be easily linked with the back-end functions to filter the queries.

8.2. Database Manger

The current database is set up in SQLite. Currently, hosting information for one chromosome is fulfilled by the database. However, in terms of scalability, if the database grows beyond or gets deployed, the best might be to switch to PostgreSQL or MySQL.

8.3. More information with the outputs

Although the user can search by gene name and alias currently, there is no indication in the output table as to whether an SNP is within a gene. In the future it would be desirable to include the associated gene name in the SNP output table, as well as have gene names and boundaries indicated on the windowed statistic graphs.

Another piece of biologically relevant information we could include in the output is the consequence of the SNP. For example, does the alternate allele result in an amino acid change? Does this have impact on the folding of a protein? Have any conditions or resistances been associated with either the reference or alternate allele?

8.4. Indels

The most consequential mutations are often indels, as these have the potential to cause frame shift in coding regions of DNA. While a single base change may change one amino acid, an indel can change every amino acid coded for after its occurrence. This usually causes large changes in the final protein's conformation and function.

SNPs involve one nucleotide being swapped for one other nucleotide, but indel mutations can result in any number of nucleotides being swapped for any number of different nucleotides. Due to this difference, the framework needed to include indel variants in our web application would likely require

different base framework than we currently have. However, since indel mutations are so transformative, it would be ideal if eventually they could be included within the web application as without them, important information is missing.

9. Contact

Janeesh Kaur Bansal: j.k.bansal@se21.qmul.ac.uk

Isabel Rachel Thompson: i.r.thompson@se21.qmul.ac.uk

Diego Pava: d.a.pava@se21.qmul.ac.uk

Aravind Pattisapu: v.pattisapu@se21.qmul.ac.uk

10. References

AUTON, A., ABECASIS, G. R., ALTSHULER, D. M., DURBIN, R. M., et al. 2015. A global reference for human genetic variation. *Nature,* 526**,** 68-74.

BEISSINGER, T. M., ROSA, G. J. M., KAEPPLER, S. M., GIANOLA, D., et al. 2015. Defining window-boundaries for genomic analyses using smoothing spline techniques. *Genetics, selection, evolution : GSE,* 47**,** 30-30.

BHATIA, G., PATTERSON, N., SANKARARAMAN, S. & PRICE, A. L. 2013. Estimating and interpreting FST: the impact of rare variants. *Genome research,* 23**,** 1514-1521.

BOOTSTRAP. 2015. *Getting started . Bootstrap .* [Online]. <http://getbootstrap.com/getting-started/#support>. [Accessed 2 March 2022].

CLARKE, L., ZHENG-BRADLEY, X., SMITH, R., KULESHA, E., et al. 2012. The 1000 Genomes Project: data management and community access. *Nature Methods,* 9**,** 459-462.

COCK, P. J. A., ANTAO, T., CHANG, J. T., CHAPMAN, B. A., et al. 2009. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics,* 25**,** 1422-1423.

DANECEK, P., BONFIELD, J. K., LIDDLE, J., MARSHALL, J., et al. 2021. Twelve years of SAMtools and BCFtools. *Gigascience,* 10.

DE JONG, M. A., WAHLBERG, N., VAN EIJK, M., BRAKEFIELD, P. M., et al. 2011. Mitochondrial DNA signature for range-wide populations of Bicyclus anynana suggests a rapid expansion from recent refugia. *PLoS One,* 6**,** e21385.

DEVANTHON-TEAM. 2021. *MySQL vs PostgreSQL vs SQlite: A comparison between 3 popular RDBMS* [Online]. https://devathon.com/blog/mysql-vs-postgresql-vs-sqlite/. [Accessed 2 March 2022].

GRINBERG, M. 2018. *Flask web development:developing web applications with python* [Online]. O' Reilly Media, Inc. [Accessed].

HARRIS, C. R., MILLMAN, K. J., VAN DER WALT, S. J., GOMMERS, R., et al. 2020. Array programming with NumPy. *Nature,* 585**,** 357-362.

HOWE, K. L., ACHUTHAN, P., ALLEN, J., ALLEN, J., et al. 2020. Ensembl 2021. *Nucleic Acids Research,* 49**,** D884-D891.

JONES, A. 2021. *Web Visualization with Plotly and Flask.* [Online]. Available: https://towardsdatascience.com/web-visualization-with-plotly-and-flask-3660abf9c946 [Accessed 3 March 2021].

JOURNALDEV. 2021. *Python io Module* [Online]. https://www.journaldev.com/19178/python-io-bytesio-stringio. [Accessed 2 March 2022].

KORUNES, K. L. & SAMUK, K. 2021. pixy: Unbiased estimation of nucleotide diversity and divergence in the presence of missing data. *Molecular Ecology Resources,* 21**,** 1359-1368.

MAGLOTT, D., OSTELL, J., PRUITT, K. D. & TATUSOVA, T. 2010. Entrez Gene: gene-centered information at NCBI. *Nucleic Acids Research,* 39**,** D52-D57.

MCKINNEY, W. 2010. *Data Structures for Statistical Computing in Python*.
MILES, A., BOT, P. I., R., M., RALPH, P., et al. 2021. cggh/scikit-allel: v1.3.3. v1.3.3 ed.: Zenodo.
PARNELL, L. D., LINDENBAUM, P., SHAMEER, K., DALL'OLIO, G. M., et al. 2011. BioStar: An Online Question & Answer Resource for the Bioinformatics Community. *PLOS Computational Biology,* 7**,** e1002216.
PLOTLY-TECHNOLOGIES-INC. 2015. Collaborative data science. Montréal, QC Plotly Technologies Inc.
PYPA. 2015. *Warehouse-pyensemble* [Online]. Available: https://github.com/pypa/warehouse [Accessed 2 March 2022].
PYTHON. 2007. *cpython - io* [Online]. https://github.com/python/cpython/blob/3.10/Lib/io.py. [Accessed 2 March 2022].
RALPH, P., THORNTON, K. & KELLEHER, J. 2020. Efficiently Summarizing Relationships in Large Samples: A General Duality Between Statistics of Genealogies and Genomes. *Genetics,* 215**,** 779-797.
STUMPF, M. P. H. 2004. Haplotype diversity and SNP frequency dependence in the description of genetic variation. *European Journal of Human Genetics,* 12**,** 469-477.
TATARINOVA, T. V., CHEKALIN, E., NIKOLSKY, Y., BRUSKIN, S., et al. 2016. Nucleotide diversity analysis highlights functionally important genomic regions. *Scientific Reports,* 6**,** 35730.
WTFORMS. 2012. *flask-wtf* [Online]. https://github.com/wtforms/flask-wtf. [Accessed 2 March 2022].