CPRISMA

Coloring Proteins By Inputs and Sets of Multiple Alignments

CPrisma User's Guide

Version 1.0

2021

Contributions from

Sergio Alejandro Poveda Cuevas

seapovedac@gmail.com

Laboratory of Computational Biophysical Chemistry

https://barrosolab.fcfrp.usp.br/

University of São Paulo

Ribeirão Preto - SP

Brazil

DESCRIPTION

CPRISMA is a bioinformatics program that gives color to multiple sequence alignment based on an input of numerical data.

LICENSE

Contents

Chapter 1

Introduction

CPRISMA (**C**oloring **PR**oteins by **I**nputs and **S**ets of **M**ultiple **A**lignments) is a command-line program designed to give color to protein multiple sequence alignments. It was written in Python language and runs on Unix systems. Since we often want to compare sets of numerical protein variables at the primary structural level, CPRISMA is an ideal program to visualize and/or detect possible differences in specific amino acids or protein regions through a color palette. CPRISMA is a versatile tool that allows the user to incorporate the features that are more convenient to highlight.

The development of this program was inspired by previous studies where we sought to highlight regions for the non-structural (NS) protein 1 from several *Flavivirus* with particular physical-chemistry or structural behavior [1, 2, 3]. We thought it would be interesting to have a capable tool to make relationships between numerical information and sequence alignments, but for any external data set.

In this documentation for CPRISMA program, we have divided the Chapters following a logical order and using practical examples to understand each parameter of the same. In Chapter 2, we will talk about the installation/uninstallation and the type of input and output (I/O) files needed to run a simulation in CPRISMA. Chapter 3 will describe the basic functions, and Chapters 4, 5, 6, and 7 more complex variables. For most examples in these six Chapters, we used NS1 proteins from Zika virus (ZIKV), Dengue variant 2 (DENV2), and West Nile viruses (WNV). Regarding ZIKV, we employ two strains from Uganda (UG) and Brazil (BR). Finally, in Chapter 8, we show three practical examples that correlate directly with three publications where we explored *i)* pK_a shifts for several NS1_{ZIKV} [1], *ii)* B-cell epitope prediction for $\text{NS1}_{\text{WNV(176–352)}}$ [2], and *iii)* the differentiation

of structural domains and biological interfaces for two ZIKV NS1 proteins [3].

Command lines, alert messages, or variables will be distinguished with the `Typewriter` font. Also, when a particular word/letter is preceded by a hyphen (-) in any tittle of this documentation it should be interpreted as a line command, otherwise, it should be interpreted as a variable.

The current developer of CPRISMA is Sergio Alejandro Poveda Cuevas at the Laboratory of Computational Biophysical Chemistry (University of São Paulo). If you have any suggestions, bug reports, or general comments about CPRISMA, please contact us at seapovedac@gmail.com.

Chapter 2

Setup

2.1 Installation/unistallation

To install CPRISMA you must directly download the code from the GitHub repositories https://github.com/seapovedac/cprisma/ or by executing the following command:

```
git clone https://github.com/seapovedac/cprisma.git
```

Then, you can invoke in that directory:

```
pip install .
```

Now your program should be in the OS environment variables hence just typing in the terminal:

```
cprisma
```[1]

or

```
python -m cprisma
```

... should return an error, where the program is requesting one of the input files for execution. If you want to be sure that there are no problems with the program, you can try to generate outputs from one of the "examples" in this directory.

To uninstall the program you can simply do it through the following command:

[1] We will use this command throughout the documentation.

```
pip uninstall cprisma
```

2.2 Inputs

To run CPRISMA, a basic knowledge of Python is required, so a good understanding of the data types that this language uses (*e.g.* sets, tuples, lists, etc.) is essential to achieve the desired results[2].

Hereafter, we will use as an example a data set of pK_a's[3] for four NS1 from ZIKV (two strains), DENV2, and WNV. Protein sequences where extracted from the NCBI (National Center for Biotechnology Information)[4], and have the next accession codes: AY632535 (ZIKV-UG), KU365777 (ZIKV-BR), PR159-S1 (DENV2), and Q9Q6P4 (WNV). We will only take into account incomplete sequences (from residues 1 to 120) and the focus will be given to the titratable residues: Asp, Glu, Arg, His, Lys, and Tyr. Cys residues are not being included, due to they are part of disulfide bridges.

CPRISMA is a program that requires at least three file kinds in the run directory[5]:

1. The alignment to put the color called "alignment.dat".
2. A CSV file called "data_input.csv".
3. A Python script called "array_get.py".

```
ZIKV-UG         DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSV
ZIKV-BR         DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSV
DENV2           DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSV
WNV             DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSV
                * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **

ZIKV-UG         SRMENIMWKSVEGELNAILEENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
ZIKV-BR         SRMENIMWRSVEGELNAILEENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
DENV2           TRLENLMWKQITPELNHILSENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKTWGK
WNV             SRLEHQMWEAVKDELNTLLKENGVDLSVVVEKQEGMYKSAPKRLTATTEKLEIGWKAWGK
                :*:*: **  :  *** :*.** *.*:::. . :.  . . . *     :*  .**:***
```

Figure 2.2.1: MUSCLE output in ClustalW format.

In addition to these files, others directories will appear (*e.g.*, "cprisma", "examples", so on), which contains all the scripts and other stuff necessary for the program. For now, let us focus on the three files mentioned above. To get the alignment data, you can directly go to the website program called

[2] See also the official Python documentation: https://docs.python.org/3.7/.
[3] pK_a's were obtained through charges of ionizable residues calculated by Monte Carlo simulations [4, 5].
[4] See https://www.ncbi.nlm.nih.gov/.
[5] These files can be found in the directory "examples/Example_0".

MUSCLE (MUltiple Sequence Comparison by Log-Expectation)[6] and run it with the FASTA format protein sequences of your interest. You can modify the input parameters as wanted, nevertheless, it is *essential* that you keep ClustalW as the output format. You can also generate these multiple sequence alignments by invoking MUSCLE through Biopython[7]. A typical MUSCLE result appears in Figure 2.2.1. All the details about this type of output is previously described [6]. An improper alignment format can produce the following output:

```
The program stop!  Improper alignment format.  Check ClustalW format at
https://www.ebi.ac.uk/Tools/msa/muscle.
```

	ZIKV-UG		ZIKV-BR		DENV2		WNV	
	A	B	C	D	E	F	G	H
1								
2	ASP	3.9	ASP	3.9	ASP	3.8899	ASP	3.7834
3	ASP	3.09737	ASP	3.38668	LYS	10.48393	ASP	3.39189
4	LYS	10.89431	LYS	10.99729	LYS	10.58842	ARG	12.78243
5	LYS	10.68626	LYS	10.69875	GLU	3.79962	GLU	3.99138
6	GLU	3.77967	GLU	3.98917	LYS	10.67679	ARG	12.39385
7	ARG	12.58017	ARG	12.2972	ASP	3.39033	HIS	6.69314
8	TYR	9.97794	TYR	9.98935	HIS	6.77441	ASP	3.58575
9	ASP	3.49226	ASP	3.79508	GLU	4.08867	GLU	4.38775
10	GLU	3.99375	GLU	4.09717	TYR	9.98403	ASP	3.48066
11	ARG	12.5932	ARG	13.38161	LYS	10.9964	ARG	12.89923
12	ASP	3.09964	ASP	2.68828	GLU	3.99943	TYR	9.79223
13	ARG	13.28718	ARG	12.97763	LYS	11.08543	LYS	11.18316
14	TYR	9.59769	TYR	9.59592	LYS	10.88321	TYR	10.19568
15	LYS	11.08461	LYS	10.99773	HIS	6.3864	TYR	9.49347

Figure 2.2.2: Data input with pK_a values for four proteins. All the information has not been showed.

Our second type of file with CSV extension is the numerical data that will be used as a basis for defining the color. Figure 2.2.2 shows an example of how this file should be. We are working with residues up to position 120, however, Figure 2.2.2 does not show all the charged amino acids up to that position. Each system is divided into 2 columns where the information on the residue kind (with code of three letters in capital) and the numerical value (in our case pK_a) will be contained.

Note: If, for example, you eliminated the pK_a value of 2.68828 *from row* 12 *and column D, CPRISMA will consider the absence of it as equal to* 0.

[6] See https://www.ebi.ac.uk/Tools/msa/muscle/.
[7] For more details: https://biopython.org/docs/1.75/api/Bio.Align.Applications.html.

```python
def array_get():

    # Tuple of residues
    target_residues = ()

    # Tuple of sequences name
    name_sequence = ()

    # Operations
    descriptor_ope = { 'n' }

    # Color
    descriptor_col = { 'nc' }

    # Visualization
    descriptor_vis = { 'ReY','DeN','LeY' }

    # Maximum restriction
    descriptor_mxr = { 'nr' }

    # Array for comparisons
    dict_ref = {}

    # Array for operations
    dict_ope = {}

    # Array for color
    dict_col = {}

    # Array for visualization
    dict_vis = {}

    # Array for maximum restriction
    dict_mxr = {}

    return target_residues, name_sequence, descriptor_ope, descriptor_col, descriptor_vis, descriptor_mxr, dict_ref, dict_ope, dict_col, dict_vis, dict_mxr
```

Figure 2.2.3: Example of how looks like the Python script "array_get.py".

Notice that there are 4 data sets for the 4 systems. In our example, the absence of any 8 columns in the CSV file can return the following error message:

```
The program stop!  Number of columns are not complete in data_input.csv.
```

On the other hand, the order as each protein appears in the alignment should match with the numerical data of the CSV file. For instance, the first dataset (highlighted by brackets) is related to the first protein (labeled as "ZIKV-UG") in the alignment sequence and so on.

Note: The program does not have the ability to distinguish to which data set each sequence belongs, so it is your responsibility to put this information in the correct order.

Finally, it is *crucial* that the first row is empty, as indicated by the arrow in Figure 2.2.2.

Note: Sometimes the numerical data length will not be the same between the systems, however, you can ignore this as long as there is a match regarding the order of appearance between the alignment and numerical data files.

The third type of file is a Python script with the variables that will be used to give different instructions to the program and obtain the desired output (Figure 2.2.3)[8].

[8]These variables and how they work will be better described in next Chapters.

2.3 Outputs

The two types of output files that CPRISMA generates are a log file called "cprisma.log" which shows relevant data from the comparisons that are made or data directly related to the input parameters, and a HTML file called "alignment.html" where the colored alignment will appear. As we will see later, due to the various features that CPRISMA has, it is better to observe with some examples how these two files work. Besides, it is worth to mention that all the warning messages or error reports will appear in the log file. Finally, you can generate other types of files with specific commands (see Section 5.14).

Chapter 3

Basic commands

To run CPRISMA, you can simply type "`cprisma`" in the terminal (for sure, at the working directory). Based on the files previously created (*i.e.*, "alignment.dat", "data_input.csv", and "array_get.py"), something you will notice is that a new log file called "cprisma.log" has been generated in your directory. In this file will appear a count of all the amino acids per system registered from "alignment.dat" (Figure 3.0.1). On the other hand, a count of total charged or hydrophobic residues will also show right away (Figure 3.0.2).

--- Number of residues in alignment ---

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	Z
ZIKV-UG	7	8	6	5	3	3	10	11	2	5	6	8	3	2	7	8	3	6	3	14	0
ZIKV-BR	7	9	6	6	3	3	9	11	2	4	6	7	3	2	7	8	3	6	3	15	0
DENV2	4	4	5	3	3	6	10	9	3	10	9	13	3	2	5	10	9	5	2	5	0
WNV	8	6	3	6	3	5	12	10	3	6	10	12	3	1	3	6	6	4	4	9	0

Figure 3.0.1: Amino acid count in log file. Note that the last column is showing the residue called "Z", this is only a way to identify those residues that do not belong to the 20 essential amino acids.

```
                        --- Kind of residues in alignment ---

ZIKV-UG charged=39 ['D', 'E', 'R', 'H', 'K', 'Y', 'C'], hydrophobic=50 ['A', 'I', 'L', 'M', 'F', 'P', 'W', 'V']
ZIKV-BR charged=39 ['D', 'E', 'R', 'H', 'K', 'Y', 'C'], hydrophobic=50 ['A', 'I', 'L', 'M', 'F', 'P', 'W', 'V']
DENV2   charged=38 ['D', 'E', 'R', 'H', 'K', 'Y', 'C'], hydrophobic=43 ['A', 'I', 'L', 'M', 'F', 'P', 'W', 'V']
WNV     charged=46 ['D', 'E', 'R', 'H', 'K', 'Y', 'C'], hydrophobic=44 ['A', 'I', 'L', 'M', 'F', 'P', 'W', 'V']

Number of columns in data_input.csv is ok!
```

Figure 3.0.2: Kind of residues count in log file.

Note: For our example, in the output showed in Figure 3.0.2 it was considered the Cys residues as charged, and this should not be. Nonetheless, CPRISMA by default counts all charged amino acids without discriminating whether or not Cys belongs to disulfide bridges.

You can see that the log file will report whether the number of columns in the CSV file is equivalent to the number of systems in the alignment file (see the last text in Figure 3.0.2).

Other additional information that will appear in the log file will be the numerical data of the CSV file (Figure 3.0.3). Although it may be redundant information because you know about that in advance, what is sought with this is to propose the user a double check of the numerical data to avoid future problems.

--- Information in data_input.csv ---

ZIKV-UG	ZIKV-UG_dat	ZIKV-BR	ZIKV-BR_dat	DENV2	DENV2_dat	WNV	WNV_dat
ASP	3.90000	ASP	3.90000	ASP	3.88990	ASP	3.78340
ASP	3.09737	ASP	3.38668	LYS	10.48393	ASP	3.39189
LYS	10.89431	LYS	10.99729	LYS	10.58842	ARG	12.78243
LYS	10.68626	LYS	10.69875	GLU	3.79962	GLU	3.99138
GLU	3.77967	GLU	3.98917	LYS	10.67679	ARG	12.39385
ARG	12.58017	ARG	12.29720	ASP	3.39033	HIS	6.69314
TYR	9.97794	TYR	9.98935	HIS	6.77441	ASP	3.58575
ASP	3.49226	ASP	3.79508	GLU	4.08867	GLU	4.38775
GLU	3.99375	GLU	4.09717	TYR	9.98403	ASP	3.48066
ARG	12.59320	ARG	13.38161	LYS	10.99640	ARG	12.89923
ASP	3.09964	ASP	2.68828	GLU	3.99943	TYR	9.79223
ARG	13.28718	ARG	12.97763	LYS	11.08543	LYS	11.18316
TYR	9.59769	TYR	9.59592	LYS	10.88321	TYR	10.19568
LYS	11.08461	LYS	10.99773	HIS	6.38640	TYR	9.49347
TYR	9.99884	TYR	9.68700	GLU	3.78878	GLU	3.99996
HIS	6.27657	HIS	6.19907	GLU	3.48118	LYS	10.69714

Figure 3.0.3: Numerical data in log file. All the information has not been showed.

Assuming both the alignment and the numerical data match, at the directory would be expected to have another new file called "alignment.html", which contains the multiple sequence alignment with color. For our example, we will not get this file, instead it is displayed the following message:

```
The program stop!  Number of target-residues [ZIKV-UG] is not the same in the input files.  Residue A: 0 (data_input.csv) != 7 (alignment.dat).
```

This error message is telling us that the number of Ala/Alanine residues (or A) for ZIKV-UG is

not the same in both the alignment and the numerical data files. This error can be corrected by applying a specific command line, but, we must know the different possible parameters that can be used through CPRISMA at first. As in other programs, when executing:

```
cprisma -h
```

... the following optional arguments will be displayed:

`-h --help`	show this help message and exit
`-v --version`	show program's version number and exit
`-ns [bool]`	rename the sequences based on a tuple
`-va [int]`	method to visualize the alignment
`-j [bool]`	join sequences (only available for `va = 2`)
`-hc [bool]`	hide the conservation line
`-n [int]`	change the first and subsequent numbers in the alignment
`-l [int]`	number of residues per line
`-tr [bool]`	get a tuple of target-residues
`-a [int]`	define the accuracy of the input data
`-t [bool]`	get all information for the different comparisons in log file
`-ck [bool]`	check reference method
`-rf [str]`	method to compare sequences (`'default'`, `'pair'`, `'multiple'`)
`-lco [bool]`	display a list of the available colors
`-ico [int]`	multiply the intensity of the color
`-sco [int]`	sequence color (when `'ssc'` descriptor is applied)
`-mco [int]`	mutation color (when `'fmac'` or `'pimc'` descriptors are applied)
`-tco [bool]`	display color at the level of 3D structure through pymol script
`-dop [bool]`	get a operation descriptors dictionary (only available for `rf = 'multiple'`)
`-dco [bool]`	get a color descriptors dictionary (only available for `rf = 'multiple'`)
`-dvi [bool]`	get a visualization descriptors dictionary (only available for `rf = 'multiple'`)
`-dmx [bool]`	get a maximum restriction descriptors dictionary (only available for `rf = 'multiple'`)

Above you can see the possible flags implemented in CPRISMA. Note that the Python variable type is appearing inside of square brackets, and a short description is displayed as well.

Next, we will try to describe each of them, bringing practical examples to better understand how

they work. Some flags should be parsed in parallel with the "array_get.py" script to avoid future misinterpretations and others may be interdependent.

The positional arguments `-h` (`--help`) and `-v (--version)` are the default commands that will show us the possible flags that can be written in the terminal and the version of the program, respectively. As these two commands are by default and self-explicit, in the following sections we will not spend time understanding them, rather it will try to focus on the CPRISMA parameters.

3.1 Target-residues (`-tr`)

As we saw previously, our problem is that the numerical data for Ala residues are missing. A possible solution could be to include in the CSV file, new rows named "ALA" followed by a column with pK_a values equal to 0 and respecting the order in which they appear for each protein.

Note: Here we consider $pK_a = 0$ as equivalent to no color, but this may vary depending on the type of data.

But this process can become time-consuming because the pK_a values for residues of Val, Ser, Trp, and so on, would also have to be included in the CSV file. A more efficient way to exclude residues that are not needed is to run the `-tr` command. However, when running it you may find the following problem:

```
The program stop!  The variable to check the target-residues is True, but the tuple is equal to 0.
```

The flag `-tr` is a Boolean command, which means that by default it is "False", but when you invoked it becomes "True". `-tr` calls the `target_residues` variable in the script "array_get.py" (Figure 2.2.3). This one is a *tuple*[1] that must contain residues in one-letter format. When `-tr` is executed, the correlation is made between the CSV file and the residues reported in the tuple. So you should try to use the same types of amino acids that appear in the numerical data file to build the tuple. Common errors that can occur in the construction of the tuple are listed below:

1. An empty tuple will return an error like the one we see above.
2. Do not put all the kind of residues that appear in the CSV file in the tuple.

[1] See https://docs.python.org/3.7/tutorial/datastructures.html#tuples-and-sequences.

3. To include a residue in the tuple that is not in the CSV file.

For our example, the tuple should be as follows:

```
target_residues = ('D', 'E', 'R', 'H', 'K', 'Y')
```

If we again run the program with the next command line:

```
cprisma -tr
```

... our log file will successfully return the following message:

```
Target-residues ('D', 'E', 'R', 'H', 'K', 'Y') match in alignment.dat and
data_input.csv.
```

Note: It is important to mention that if you have the numerical values of all residues that appear in a specific multiple sequence alignment, it is not necessary to explicitly define a tuple with the 20 essential amino acids in the script "array_get.py" because the program already defines those 20 residues by default (see the practical example of Section 8.3*).*

3.2 Accuracy (-a)

As we will see later, the color may not have enough intensity depending on the numerical data we use, due to the limits that the HTML color code itself may have, or because the limits of the human vision. Sometimes it will be necessary to make additional adjustments to improve the intensity to highlight the regions that matter most to us in the sequence alignment.

In this section, let us focus on the accuracy of the data, which is changeable by means of `-a` command[2]. In Figure 3.0.3, it is observed that the precision of the data goes up to the fifth decimal place. As showed in Figure 3.2.1, after program processing one decimal per value will be displayed (which corresponds to the default condition). If you want to consider 2 or more decimals, it can invoke the command `-a` followed by an positive integer number which refers to the decimal places, as follows:

```
cprisma -tr -a 3
```

[2]This is one of the possible methods to improve the color intensity. Other methods can also be applied (see Section 5.15 and Chapter 7), but we will see them later.

In our example, we are ordering the program to only consider 3 decimal places. Remember that this command will only affect the posible values obtained from operations, and then, the color applied when a gradient is executed (see Chapters 4, 5, and 6, respectively). Nevertheless, some information in the log file will not be affected by this parameter, like the statistical data reported there (Figure 3.2.2).

--- Data processed ---

	ZIKV-UG	ZIKV-UG_dat	ZIKV-BR	ZIKV-BR_dat	DENV2	DENV2_dat	WNV	WNV_dat
0	ASP	3.9	ASP	3.9	ASP	3.9	ASP	3.8
1	ASP	3.1	ASP	3.4	XXX	0.0	ASP	3.4
2	XXX	0.0	XXX	0.0	LYS	10.5	XXX	0.0
3	LYS	10.9	LYS	11.0	XXX	0.0	ARG	12.8
4	LYS	10.7	LYS	10.7	LYS	10.6	XXX	0.0
5	GLU	3.8	GLU	4.0	GLU	3.8	GLU	4.0
6	ARG	12.6	ARG	12.3	LYS	10.7	ARG	12.4
7	TYR	10.0	TYR	10.0	XXX	0.0	HIS	6.7
8	XXX	0.0	XXX	0.0	ASP	3.4	XXX	0.0
9	ASP	3.5	ASP	3.8	XXX	0.0	ASP	3.6
10	GLU	4.0	GLU	4.1	HIS	6.8	GLU	4.4
11	ARG	12.6	ARG	13.4	XXX	0.0	XXX	0.0
12	ASP	3.1	ASP	2.7	GLU	4.1	ASP	3.5
13	ARG	13.3	ARG	13.0	XXX	0.0	ARG	12.9
14	TYR	9.6	TYR	9.6	TYR	10.0	TYR	9.8
15	LYS	11.1	LYS	11.0	LYS	11.0	LYS	11.2
16	TYR	10.0	TYR	9.7	XXX	0.0	TYR	10.2
17	HIS	6.3	HIS	6.2	XXX	0.0	TYR	9.5

Figure 3.2.1: Numerical data processed in log file. Note that some rows are named "XXX", which indicates that a gap has been inserted. These gaps are directly related to the alignment and are necessary for the reading and coloring of the sequences later. Notice that there is an index starting from 0 that number each row. All the information has not been showed.

--- Statistics ---

	ZIKV-UG	ZIKV-BR	DENV2	WNV
count	36.000000	36.000000	35.000000	43.000000
mean	8.027069	8.065460	7.998239	7.820189
std	3.957723	4.020073	3.578109	3.732822
min	3.097370	2.688280	3.390330	3.384640
25%	3.964313	3.990678	3.892795	3.993175
50%	9.787815	9.641460	9.984030	9.493470
75%	11.078850	11.322598	10.883975	11.034720
max	13.392970	13.485050	13.184100	13.491570

Figure 3.2.2: Statistical data for each protein in log file.

3.3 Alignment visualization (-va)

At this point, you will have noticed that the file "alignment.html" appears in your directory. The output that was obtained for our alignment appears in Figure 3.3.1. See that the file does not have any color still[3]. As you can observe, by default, CPRISMA will separates each protein sequence keeping the conservation line for each of them [*i.e.*, the line that contains dots (".", ":"), asterisks ("*") or spaces (" ")]. Now it seems not to have much relevance to observe our multiple sequence alignment without comparisons among sequences. Nevertheless, as you will see later this kind of visualization can be very useful to detect independent color behaviors from sets of numerical data.

```
        1                 20                  40                  60
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE
        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                 120
ZIKV-UG ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
        ** *.*:::. . :.  . . . *     :*  .**:***

        1                 20                  40                  60
ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE
        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                 120
ZIKV-BR ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
        ** *.*:::. . :.  . . . *     :*  .**:***

        1                 20                  40                  60
DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS
        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                 120
DENV2   ENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKTWGK
        ** *.*:::. . :.  . . . *     :*  .**:***

        1                 20                  40                  60
WNV     DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK
        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                 120
WNV     ENGVDLSVVVEKQEGMYKSAPKRLTATTEKLEIGWKAWGK
        ** *.*:::. . :.  . . . *     :*  .**:***
```

Figure 3.3.1: Alignment processed through visualization method 1 without sequence comparison.

Another way to visualize our alignment can be calling the method 2 through the `-va` command, as follows:

[3] This aspect will be covered in Chapter 5.

```
cprisma -tr -va 2
```

An output using this method appears in Figure 3.3.2. Notice that each sequence is being labeled at the end with the word "comparison" followed by a number (see brackets in Figure 3.3.2). This means that CPRISMA is not comparing the sequences but only analyzing each protein separately, as we saw in the previous method.

Note: We can apply these same visualization methods but comparing our sequences. To see this aspect in more detail it is recommended to check Sections 3.9 and 3.10.

```
        1                 20                 40                 60
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE  comparison 1

ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE  comparison 2

DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS  comparison 3

WNV     DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK  comparison 4

        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                120
ZIKV-UG ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK  comparison 1

ZIKV-BR ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK  comparison 2

DENV2   ENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKTWGK  comparison 3

WNV     ENGVDLSVVVEKQEGMYKSAPKRLTATTEKLEIGWKAWGK  comparison 4

        ** *.*:::. . :.  . . . *     :*  .**:***
```

Figure 3.3.2: Alignment processed through visualization method 2 without sequence comparison.

3.4 Join (`-j`)

The `-j` command is a complement for the visualization method 2. With this what we want is to reduce the space between the comparisons to come from the visualization in Figure 3.3.2 to the outcome shown in Figure 3.7.1. To get an alignment without spaces, you can invoke the following command:

```
cprisma -tr -va 2 -j
```

Practical examples using this flag appears in Chapter 8.

3.5 Numbering (-n)

To change the first and subsequent numbers that appears in the top of the alignment, the command `-n` can be executed followed by a positive integer number corresponding to the desired position. For instance, we can assume that our first number is 101, so:

```
cprisma -tr -va 2 -j -n 101
```

See the practical example of Section 8.2.

3.6 Number of residues per line (-l)

You can change the number of amino acids per line using the command `-l` followed by a positive integer number.

Note: Remember that 80 residues per line will appear by default, so we recommend only use values greater than and multiples of 80 to have an optimal display of the numbering. You can use whatever amount of residues per line you prefer, but there may be times when the numbers will be cut off.

At next, an example of a command line for this feature appears:

```
cprisma -tr -va 2 -j -l 160
```

3.7 Name sequence (-ns)

Something that is useful when working with different sequences is renaming them. For this, you can run the command `-ns`. As we saw with `-tr`, this command that renames the sequences calls the variable `name_sequence` of the script "array_get.py" (Figure 2.2.3) and makes a relationship between this variable (which is a tuple) and the number of proteins analyzed. Let us imagine that we build the following tuple:

```
name_sequence = (“Sequence 1”, “Sequence 2”, “Sequence 3”)
```

Then, it is executed the next command line:

```
cprisma -tr -va 2 -j -ns
```

Since the number of "name sequences" is only 3 and the total of proteins is 4, CPRISMA will return the following error:

```
Variable check_name_seq = True but the sequences' number does not match with tuple input ('Sequence 1', 'Sequence 2', 'Sequence 3')
```

Adding one more name to the tuple `('Sequence 1', 'Sequence 2', 'Sequence 3', 'Sequence 4')`, the program will finally return the following message:

```
Variable check_name_seq = True, so the sequences' names have been changed.
```

Now you can see that both your alignment (Figure 3.7.1) and all the information in the log file have changed the sequences' names.

```
                1                 20                 40                 60
Sequence 1      DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE  comparison 1
Sequence 2      DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE  comparison 2
Sequence 3      DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS  comparison 3
Sequence 4      DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK  comparison 4
                * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

                81                100                120
Sequence 1      ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK  comparison 1
Sequence 2      ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK  comparison 2
Sequence 3      ENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKTWGK  comparison 3
Sequence 4      ENGVDLSVVVEKQEGMYKSAPKRLTATTEKLEIGWKAWGK  comparison 4
                ** *.*:::. . :.  . . . *     :*  .**:***
```

Figure 3.7.1: Alignment with renamed sequences.

3.8 Hide conservation line (`-hc`)

The `-hc` command hide the conservation line of the alignment (check the line with "*", ".", ":" and spaces " " in Figure 3.7.1, for instance). To hide that line, you can invoke the following command:

```
cprisma -tr -hc
```

Hiding the conservation line can be useful when we are using transcendental comparison methods (see Section 3.10).

See the practical example of Section 8.2 where `-hc` is employed.

3.9 Check reference method (`-ck`)

So far you have surely noticed the following message in your log file:

```
No comparison among the data.
-- Set of comparisons:  main_0 --
     ZIKV-UG X no comparison
-- Set of comparisons:  main_1 --
     ZIKV-BR X no comparison
-- Set of comparisons:  main_2 --
      DENV2 X no comparison
-- Set of comparisons:  main_3 --
       WNV X no comparison
```

For our example, the data of the protein systems are not being compared. However, it can be interesting to make that because you could later apply mathematical operations (see Chapter 4). For instance, in our previous study, we wanted to compare the pK_a's of ionizable residues among NS1's proteins from several ZIKV strains [1]. For that, it was calculated the difference of pK_a's (ΔpK_a) among proteins and they were visualized with specific colors (see Section 5.10 and the practical example described in Section 8.1 as well).

Before to do any calculation, you must inform the program that it is desired to compare the data by means of the `-ck` flag. The command line to execute comparisons between the protein sequences appears below:

```
cprisma -tr -ck
```

Now, the following information will be found in the log file:

```
-- Set of comparisons:  main_0 --

   ZIKV-UG X ZIKV-BR DENV2 WNV
```

```
           1                  20                  40                  60
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE
ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE
DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS
WNV     DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK
           * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

           81                  100                 120
ZIKV-UG ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
ZIKV-BR ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
DENV2   ENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKTWGK
WNV     ENGVDLSVVVEKQEGMYKSAPKRLTATTEKLEIGWKAWGK
           ** *.*:::. . :.  . . . *     :*  .**:***
```

Figure 3.9.1: Alignment processed through visualization method 1 with sequence comparison.

Note that now the sequences are being compared, so ZIKV-UG is the reference sequence, and ZIKV-BR, DENV2, and WNV are the target-sequences. Also, this comparison is being coupled to a group of comparisons called "`main_0`". This is not the same as what was seen previously where different "`main`'s" were generated. At the same time, you will have perceived that your HTML file (Figure 3.9.1) is different when compared to the previous outcomes (Figures 3.3.1 and 3.3.2). Technically, the alignment is the same that we use as input (Figure 2.2.1) but with a slightly different format.

Note: In Section 3.10, you will find more information on how to apply more transcendental comparisons.

3.10 Reference method (`-rf`)

If `-ck` was executed you will have other additional information in your log file:

```
Method used 'default' for the array:  | main_0 [{'0':  [1, 2, 3]}] |
```

To understand what this means, you need a basic understanding of Python's data structure. Let

us go a little back, just to correctly interpret these outcomes. Based on the previous message, note that you have one comparison that belongs to `main_0`. In Python, whenever one string type (II) and any type variables (VI) appear separated by a colon (V) inside brackets (I), it will be interpreted by the language as a *dictionary*[4] (Figure 3.10.1) . The numerals II and VI are also called *keys* and *values*, respectively. In our case, the key of our dictionary is the string '0' and the value is a *list* of numbers from 1 to 3. Note that the lists have square brackets (III), which differentiates them from dictionaries[5].

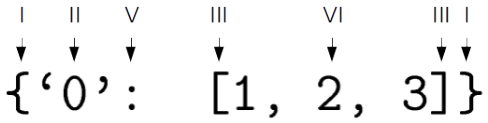


Figure 3.10.1: Dictionary created by CPRISMA. Meaning of Roman numerals is explained in the text.

Note: Based on the message above, see that your dictionary is also inside square brackets: `[{'0': [1, 2, 3]}]`[6]*. In the Subsection 3.10.3, we will explain in more detail the reason for that.*

Please keep all these features in mind as they will be helpful in understanding the following sections. Furthermore, in some moments we will call the sets of comparisons as *arrays*.

To interpret the numbers that is in our dictionary (Figure 3.10.1), we simply have to relate the order of appearance for each sequence based on the multiple alignment. Thus, '0' will be ZIKV-UG, '1' will be ZIKV-BR, and so on.

There are three comparison methods used by CPRISMA: `default`, `pair` and `multiple`, which are described below.

3.10.1 Default

The `default` method of CPRISMA compares the first sequence with the rest. You can explicitly invoke this method via the `-rf` flag followed by the word "default" like this:

[4] See https://docs.python.org/3.7/tutorial/datastructures.html#dictionaries.
[5] See https://docs.python.org/3.7/tutorial/datastructures.html#more-on-lists.
[6] We have changed the color to purple for some square brackets to distinguish easily each list.

```
cprisma -tr -ck -rf default
```

Note: Remember, due to this comparison method is the optional argument by default (as its name implies), CPRISMA will always return a dictionary with this structure when you run `-ck` *(i.e., you do not need to type* `-rf default`*, explicitly)*

Comparison: 1

```
        1                 20                 40                 60
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE
ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE
        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                120
ZIKV-UG ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
ZIKV-BR ENGVQLTVVVGSVKNPMWRGPQRLPVPVNELPHGWKAWGK
        ** *.*:::. . :.  . . . *     :*  .**:***
```

Comparison: 2

```
        1                 20                 40                 60
DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS
WNV     DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK
        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.

        81                100                120
DENV2   ENEVKLTIMTGDIKGIMQAGKRSLRPQPTELKYSWKTWGK
WNV     ENGVDLSVVVEKQEGMYKSAPKRLTATTEKLEIGWKAWGK
        ** *.*:::. . :.  . . . *     :*  .**:***
```

Figure 3.10.2: Alignment processed with `pair` comparison method.

3.10.2 Pair

Another comparison method is called `pair`. To invoke it, we can type in terminal:

```
cprisma -tr -ck -rf pair
```

Now, our log file will contain the following information:

```
Method used 'pair' for the array:  | main_0 [{'0':  [1]}] | | main_2 [{'2':  [3]}] |

-- Set of comparisons:  main_0 --
```

```
        ZIKV-UG X ZIKV-BR

-- Set of comparisons:  main_2 --

           DENV2 X WNV
```

Note that two arrays have now been created, which can be interpreted as two independent sets of comparisons and they belong to the `main`'s: `main_0` and `main_2`. One where 0 (ZIKV-UG) and 1 (ZIKV-BR) sequences are the reference$_{\text{comparison1}}$ and target$_{\text{comparison1}}$, respectively; and another where 2 (DENV2) and 3 (WNV) sequences are the reference$_{\text{comparison2}}$ and target$_{\text{comparison2}}$, respectively. This will also be reflected in our HTML file (Figure 3.10.2)[7].

It is important to mention that for CPRISMA to work correctly with the `pair` comparison method, the total number of sequences is needed to be *even*. Otherwise, it will return the method for `default` with the following message:

```
Attention!!!  Method selected by user is 'pair' but number of sequences (=3[8]) is
not a even number.  The method return to 'default'.
```

3.10.3 Multiple

Among all the comparison methods, `multiple` can be the most complex to understand. With this method, many types of comparisons can be made between a set of sequences (*e.g.* see all the practical examples in Chapter 8). However, before invoking it you must have built a dictionary compatible with the CPRISMA format array. In the script "array_get.py" a variable called `dict_ref` will appear (Figure 2.2.3). From this, you can build an array like this:

```
dict_ref = { 'main_0' :  [ {'0':[1, 2, 3]}, {'0':[]} ] ,

             'main_1' :  [ {'1':[0, 2]}, {'1':[0, 2, 3]} ] ,

             'main_2' :  [ {'2':[0]} ] }
```

At first glance, this type of arrangement may seem tricky, but give us a chance to explain it to make it easier.

[7] We invite the user to also explore visualization method 2 (see Section 3.3).
[8] A data set of 3 proteins was used to compare them (only as an example).

The CPRISMA convention is going to call the "external layer" of the array with the names: "*first-key*" (or "*main key*") and "*first-value*" (or "*main value*"). Then, the first-keys will be all `main`'s typed in the array, *i.e.*, `main_0`, `main_1`, and `main_2`; and the first-values will be the lists: `[{'0': [1, 2, 3]}, {'0': []}]`, `[{'1': [0, 2]}, {'1': [0, 2, 3]}]`, and `[{'2': [0]}]`[9]. Each element of the list/first-value is equivalent to a comparison and they has a "*second-key*" (or "*subkey*") and a "*second-value*" (or "*subvalue*"). The subkeys are the string numbers that represent the sequence references and the subvalues are the lists with the target proteins. Notice that each cluster of first-key/first-value pairs is separated by a comma and they are so-called as set of comparisons[10]. Besides, each comparison in the list/first-value is separated by a comma as well. Please, avoid confusing this! For the sake of convenience, in Figure 3.10.3, we have made a sketch for `main_0`, where you can follow each characteristic of the CPRISMA array.

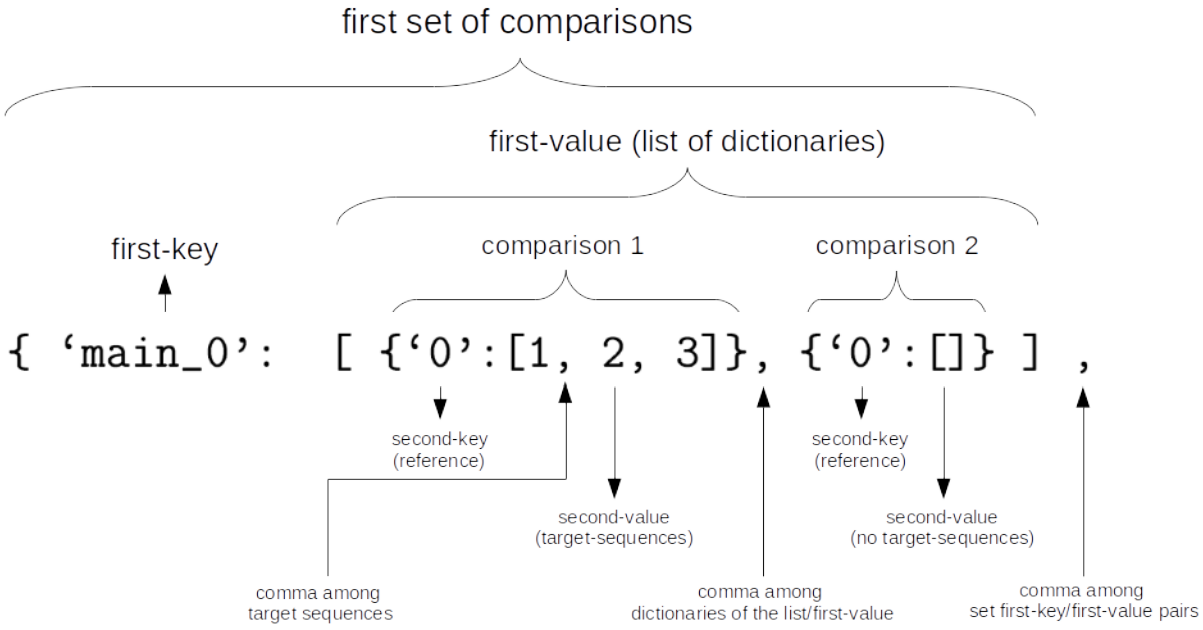


Figure 3.10.3: Example of the structure of the CPRISMA array when is used `multiple` method.

Maybe, you can also perceive that each `main` has the same reference number of a particular set of comparisons. For example, `main_1` has comparisons 3 and 4 which are `{'1': [0, 2]}` and `{'1':`

[9] We have changed the color to purple for some square brackets to distinguish easily each list.
[10] Identify this same structure when using the `default` and `pair` methods.

`[0, 2, 3]}`, respectively, so the reference sequences `'1'` match with the number of the `main` (*i.e.*, `main_1`). The same goes for `main_0` and `main_2`. However, it should be noted that you can use whatever name you prefer for each "`main`", to better identify your comparisons. Just make sure they are different from each other. For practical purposes, we will use the standard name as already seen in the previous and following examples.

Note: Check that comparison 2 has a list with no elements or empty: `{'0': []}`. *CPRISMA will interpret this dictionary as "without comparison" (or without target-sequences).*

If the array is well constructed, we can invoke the following command:

```
cprisma -tr -ck -rf multiple
```

Then, in our log file the comparisons will appear as follows:

```
-- Set of comparisons:  main_0 --

   ZIKV-UG X ZIKV-BR DENV2 WNV

     ZIKV-UG X no comparison

-- Set of comparisons:  main_1 --

     ZIKV-BR X ZIKV-UG DENV2

   ZIKV-BR X ZIKV-UG DENV2 WNV

-- Set of comparisons:  main_2 --

          DENV2 X ZIKV-UG
```

Our HTML file will show an output like the one shown in Figure 3.10.4 (to facilitate the visualization, the method 2 was used, it is recommended to see Section 3.3).

Finally, if the array is not well built, CPRISMA will always return to the `default` value. In addition, it will inform you in detail where the errors are. Based on this, you can correct them yourself (the program will guide you). Warning/error messages for `multiple` command will not appear in this documentation, as we prefer that the user be trained in understanding the construction of CPRISMA arrays.

See in Chapter 8 the particulars that can be declared in `dict_ref` using `multiple` comparison method .

```
        1                 20                  40                  60
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE  comparison 1
ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE
DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS
WNV     DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK

ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE  comparison 2

ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE  comparison 3
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE
DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS

ZIKV-BR DVGCSVDFSKKETRCGTGVFVYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEDGICGISSVSRMENIMWRSVEGELNAILE  comparison 4
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE
DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS
WNV     DTGCAIDISRQELRCGSGVFIHNDVEAWMDRYKYYPETPQGLAKIIQKAHKEGVCGLRSVSRLEHQMWEAVKDELNTLLK

DENV2   DSGCVVSWKNKELKCGSGIFITDNVHTWTEQYKFQPESPSKLASAIQKAHEEGICGIRSVTRLENLMWKQITPELNHILS  comparison 5
ZIKV-UG DVGCSVDFSKKETRCGTGVFIYNDVEAWRDRYKYHPDSPRRLAAAVKQAWEEGICGISSVSRMENIMWKSVEGELNAILE

        * ** :. ..:* .**:*:*: ::* :* :.**: *::*  **  :::*.::*:**: **:*:*: **  :  *** :*.
```

Figure 3.10.4: Alignment processed with `multiple` comparison method. Not all alignment is being shown.

3.11 Total information (`-t`)

If you want to have the verbose version of your log file, you can invoke the following command:

```
cprisma -tr -t
```

Now, the log file will show in detail the operations that you are applying to each set of comparisons (see Chapter 4). In turn, a relationship between each comparison and the color assigned to it will be shown (see Chapter 5). Also, if a maximum restriction is applied (see Chapter 7), you can see the lists of values used to normalize the color displayed on the HTML file (whether or not math operations are performed).

Chapter 4

Operation (`descriptor_ope`)

CPRISMA has a feature called "operation" that allows the user to computed basic mathematical operations between the data. To execute this feature, it is *mandatory* to have at least one comparison between particular reference and target proteins. For the following examples, we will always assume the `default` comparison method (see Subsection 3.10.1), with the exception of Section 4.5.

Generally, the program always will read the variable called `descriptor_ope` from the Python script "array_get.py" (Figure 2.2.3). Then, the CPRISMA will perform a certain operation between the vector $\mathbf{r} = [r_j]$ (reference) and the matrix $\mathbf{T} = [t_{ij}]$ [target(s)]. Based on the data processed and displayed in the log file, for our example using pK_a values, the vector $\mathbf{r}$ and matrix $\mathbf{T}$ have dimensions 56×1 and 56×3, respectively. Something that should be noted is that CPRISMA takes advantage of the `numpy` features to apply broadcasting and thus improve the speed of matrix operations. For this version of CPRISMA, you can perform 3 types of calculations: delta/`d` (Δ), absolute delta/`da` ($|\Delta|$) , and multiplication/`m`. However, it is possible to do not consider any operation (none/`n`) as well. For CPRISMA these 4 possibilities are called *descriptors*.

Note: We highlight that this nomenclature (i.e., descriptor) also applies to other features in addition to operations, such as color, visualization, and maximum restriction (see Chapters 5, 6, *and* 7, *respectively).*

Finally, the numerical data generated through the operations will serve as the basis to put the color in the multiple sequence alignments.

4.1 None (n)

If it is desired to keep the same numerical values of the input (*i.e.*, the data of the CSV file), you can do this by declaring the descriptor `n` in the "array_get.py" file, as follows:

```
descriptor_ope = { 'n' }
```

You can perceive that your `n` descriptor is a string variable inside a pair of brackets. This data structure corresponds to a *set* variable type and should not be confused with a dictionary[1].

After having run CPRISMA, your log file will have various information about the operation applied. For now, we should only focus on the next[2]:

main	comparison	descriptor
main_0	1	n

This statement is simply saying that `n` descriptor was considered for comparison 1, in the `main_0` comparison set.

Note: Sometimes, it could be declared this descriptor or another wrongly. To avoid that the program stops, it will always return to `n` *descriptor (default parameter), with some additional error messages in your log file, which will guide you to correct the problem.*

4.2 Delta (d)

When the `d` descriptor is invoked like this:

```
descriptor_ope = { 'd' }
```

... the program will do a conventional Δ calculation between $\mathbf{r}$ and $\mathbf{T}$ (Equation 4.2.1).

$$\mathbf{W} = \mathbf{T} - \mathbf{r}, \tag{4.2.1}$$

where $\mathbf{W}$ is the new data to put color (see Chapter 5).

[1] See https://docs.python.org/3.7/tutorial/datastructures.html#sets.
[2] When using a different operation descriptor, respectively, its letter will appear in the `descriptor` column.

4.3 Absolute delta (da)

To declare the `da` descriptor, you can do it as follows:

```
descriptor_ope = { 'da' }
```

CPRISMA will apply the Δ calculation as before (Section 4.2), but using the absolute value:

$$\mathbf{W} = |\mathbf{T} - \mathbf{r}| \tag{4.3.1}$$

4.4 Multiply (m)

To declare the `m` descriptor, it is possible as follows:

```
descriptor_ope = { 'm' }
```

The program will multiply the data as showed in Equation 4.4.1.

$$\mathbf{W} = \mathbf{T} \times \mathbf{r} \tag{4.4.1}$$

4.5 Operation descriptors dictionary (-dop)

One can run the following command using `multiple` comparison method:

```
cprisma -tr -ck -rf multiple
```

We have started from the next CPRISMA array (see Subsection 3.10.3)[3]:

```
dict_ref = { 'main_0':  [ {'0':[1, 2, 3]}, {'0':[]} ] ,
             'main_1':  [ {'1':[0, 2]}, {'1':[0, 2, 3]} ] ,
             'main_2':  [ {'2':[0]} ] }
```

Suppose we are doing a Δ operation (*i.e.*, `descriptor_ope = { 'd' }`). So you will have noticed the following information in your log file:

[3] Note that we have differentiated each operation descriptor for each comparison with a different color to aid the reader.

```
Comparing dict_ref with dict_ope:

                                {'0':  [1, 2, 3]} ........  d

                                {'0':  []} ........  d

                                {'1':  [0, 2]} ........  d

                                {'1':  [0, 2, 3]} ........  d

                                {'2':  [0]} ........  d

The array of the feature 'operation' dict_ope is compatible with the array of
comparison sequences dict_ref!
```

Note: The comparison `2` *does not have a target-sequence (i.e.,* `{'0':[]}`*), so operations like* `d`, `da`, *or* `m` *are not applied.*

This information is showing us a relationship between each comparison and the type of operation that we are applying to each one. For all comparisons, a Δ is computed among its specific reference and target-sequences.

However, we are not always going to calculate a single operation for all comparisons. To perform a different math calculation for each reference/target(s) relationship, an array of operation descriptors called `dict_ope` should be built in our "array_get.py" script. We shall follow very similar rules to those we saw earlier for the `multiple` method. Our dictionary of operations must have an equivalent number of `main`'s as well as in `dict_ref` (*i.e.*, `main_0`, `main_1`, and `main_2`). In turn, for each `main` (first-key) we should have a list of operation descriptors (first-values)[4]. Based on this, suppose we construct the following operation-array[5]:

```
dict_ope = { 'main_0' :  [ {'d'} , {'n'} ] ,

             'main_1' :  [ {'m'} , {'d'} ] ,

             'main_2' :  [ {'da'} ] }
```

Invoking the `-dop` command you will order to CPRISMA to take into account the `dict_ope` dictionary of the "array_get.py" script:

[4] As we will see later for the other features (*i.e.*, color, visualization, and maximum restriction), these first-values will always be a list of descriptors with the same number of elements equivalent to the number of comparisons.

[5] Note that we have differentiated each operation descriptor for each comparison with a different color to aid the reader.

```
cprisma -tr -ck -rf multiple -dop
```

Now, our log file will show the operation applied for each comparison:

```
Comparing dict_ref with dict_ope:

                                {'0':  [1, 2, 3]} ........  d

                                {'0':  []} ........  n

                                {'1':  [0, 2]} ........  m

                                {'1':  [0, 2, 3]} ........  d

                                {'2':  [0]} ........  da

The array of the feature 'operation' dict_ope is compatible with the array of
comparison sequences dict_ref!
```

... and the next table:

main	comparison	descriptor
main_0	1	d
main_0	2	n
main_1	3	m
main_1	4	d
main_2	5	da

It is important to mention that `-dop` will only work with the `multiple` comparison method.

Note: If you want to check how the numerical calculations are applied between each row of the processed input file, you can use the `-t` *flag (see* Section 3.11*).* Figure 7.0.1 *shows an example of how looks like that part of the log file.*

Note: If the array of `dict_ope` *is wrongly built, all descriptors will be transformed to* `n` *(default parameter) for each comparison and additional information about where the errors are will be displayed.*

Chapter 5

Color (`descriptor_col`)

So far, we have intentionally seen situations where our multiple sequence alignment is colorless. This is because we did not want to work with many features simultaneously. In this Chapter, it will be described how to use the CPRISMA feature called "color".

Remember that the color applied is using as a base the HTML language. The file used by CPRISMA to apply the color is located at "cprisma/colors.csv". Figure 5.0.1 shows some rows from this file. You can see that the cells are divided into four columns and the first row shows heads as: General, Name, Hex, and RGB. CPRISMA will only use the column labeled RGB to give color, the other information is just to help the user to identify each one easily. It is important to mention that all this stuff is based on the official HTML site about color code[1]. For quick access to this list of colors you can invoke the `-lco` command described in Section 5.16.

	A	B	C	D
1	General	Name	Hex	RGB
2	Red HTML	IndianRed	#CD5C5C	rgb(205, 92, 92
3	Red HTML	LightCoral	#F08080	rgb(240, 128, 128
4	Red HTML	Salmon	#FA8072	rgb(250, 128, 114
5	Red HTML	DarkSalmon	#E9967A	rgb(233, 150, 122
6	Red HTML	LightSalmon	#FFA07A	rgb(255, 160, 122
7	Red HTML	Crimson	#DC143C	rgb(220, 20, 60
8	Red HTML	Red	#FF0000	rgb(255, 0, 0

Figure 5.0.1: CSV file with color information. All the rows has not been showed.

[1] See https://htmlcolorcodes.com/color-names/.

Just as the operation feature has a variable declared in the script "array_get.py", for color feature we also have a set variable called `descriptor_col` (Figure 2.2.3). For this, you will have 10 different descriptors (`nc`, `ssc`, `fsc`, `fac`, `fmac`, `pic`, `pimc`, `tc`, `tmc`, and `pkac`) and they will be described below.

Finally, for most next examples the following command on terminal and variable in your Python script "array_get.py" is being executed:

```
cprisma -tr -ck
```

```
descriptor_ope = { 'd' }
```

5.1 No color (nc)

To use the descriptor `nc`, you can declare it like this:

```
descriptor_col = { 'nc' }
```[2]

As the name implies, you are not adding any color to the alignment. Now, your log file will have various information about the color feature applied. We should only focus on the next:

	main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0		1	nc	NaN	NaN	NaN	NaN	NaN

This table is simply saying that `nc` descriptor was considered for comparison 1, in the `main_0` comparison set (for now ignore the rest of the columns in that log message).

Note: Sometimes, it could be declared this descriptor or another wrongly. To avoid that the program stops, it will always return to `nc` *descriptor (default parameter), with some additional error messages in your log file, which will guide you to correct the problem.*

5.2 Same sequence color (ssc)

Suppose that we want to give the same color to a specific set of sequences. For this we declare:

[2] For all alignment examples of Chapter 3, this was the variable executed.

```
descriptor_col = { 'ssc' }
```

The log file will report as `[same]` in the `sequence` column:

	main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0		1	ssc	[same]	NaN	NaN	NaN	NaN

Furthermore, your HTML file now has a colored alignment (Figure 5.2.1).

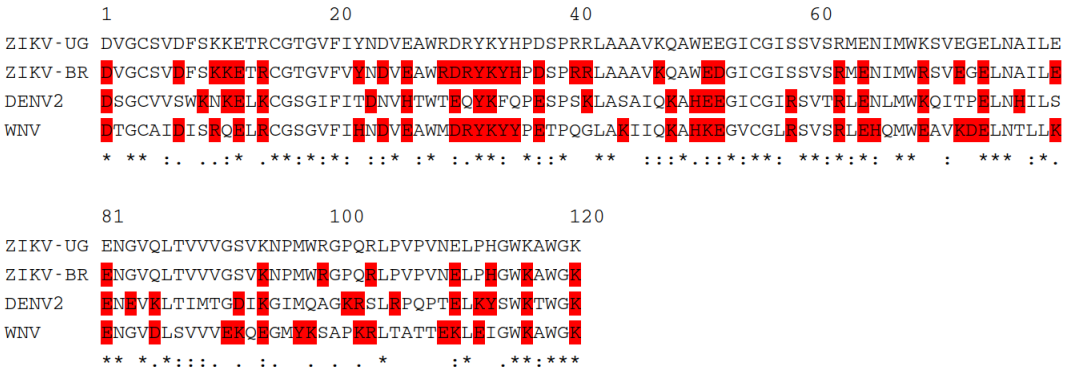


Figure 5.2.1: Alignment colored through `ssc` descriptor.

Note that only the target-residues in all the proteins are being colored with Red (see Section 3.1) and the reference (*i.e.*, ZIKV-UG) is not being considered. The default color for the `ssc` descriptor is Red, which corresponds to the 6th row when executed `-lco` (see Section 5.16). You can change that color by invoking the `-sco` flag described in Section 5.12.

5.3 Free sequence color (`fsc`)

If we want to give a different color to each *target-sequence*, we can invoke the descriptor `fsc` as follows:

```
descriptor_col = { 'fsc' :  [6, 54, 94] }
```

Notice that now, what is declared is not a set variable type but a dictionary. Our key will be the descriptor `fsc` (string type) and our value is a list of number colors (see Section 5.16) that

corresponds to the total number of target-sequences and their order of appearance (*i.e.*, colors `6`, `54` and `94` for ZIKV-BR, DENV2 and WNV, respectively). If it is run the program, our log file will show the following information:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	fsc	[6, 54, 94]	NaN	NaN	NaN	NaN

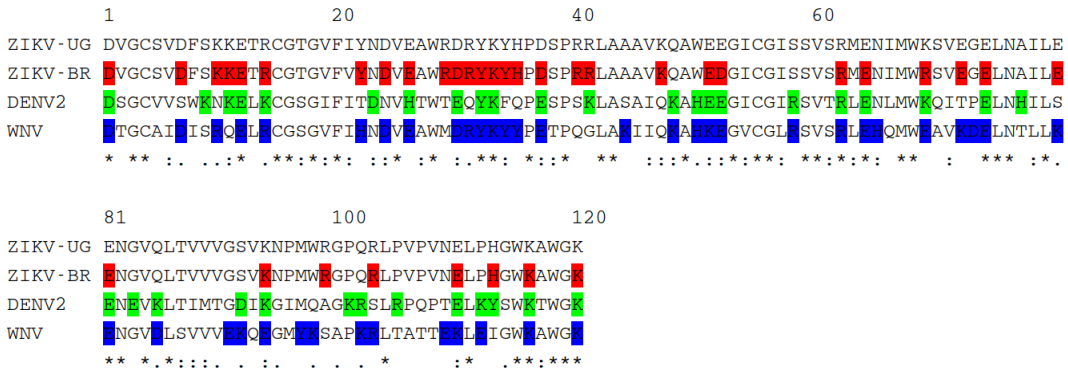


Figure 5.3.1: Alignment colored through `fsc` descriptor.

... and the HTML file will now display a color for each target-sequence (Figure 5.3.1). Assuming you will do quick tests and do not care about the color kind assigned to each target-sequence, it is possible to declare your variable like this:

```
descriptor_col = { 'fsc' :  ['random'] }
```

If the word `'random'` is typed into the list, CPRISMA will give each target-sequence a random color.

5.4 Free amino acid color (`fac`)

You can also color each *target-residue* for the target-sequence(s) like this:

```
descriptor_col = { 'fac' :  [6, 20, 22, 54, 94, 46] }
```

Look at the list has 6 different color numbers. That number of elements is correlated to the number of target-residues declared in the tuple `target_residues` of the script "array_get.py" (see Section 3.1).

Note: If you need to consider the 20 *essential amino acids, you will need to declare* 20 *color numbers in that list.*

If you run CPRISMA program, the log file will report this list of color numbers in the `residue` column like this:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	fac	NaN	[6, 20...]	NaN	NaN	NaN

... and the HTML file will now display a color for each target-residue (Figure 5.4.1). As with `fsc` descriptor, you can declare random colors for your target-residues as follows:

```
descriptor_col = { 'fac' :  ['random'] }
```

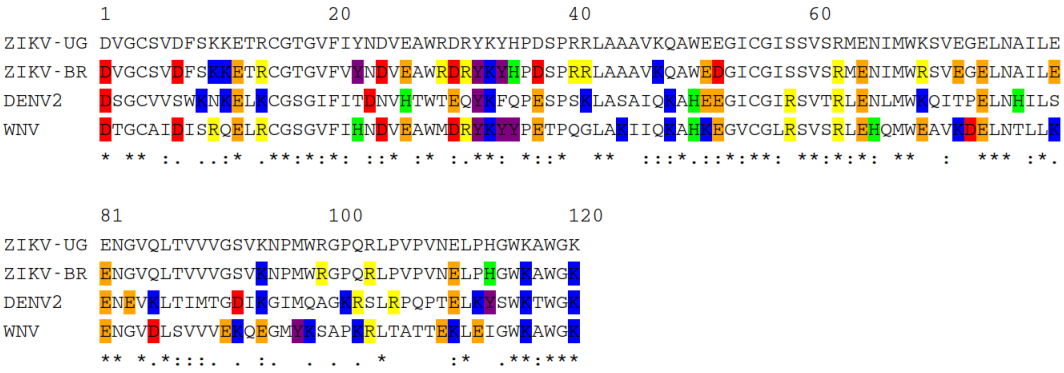


Figure 5.4.1: Alignment colored through `fac` descriptor.

5.5 Free amino acid and mutation color (`fmac`)

The `fmac` descriptor is technically a copy of the `fac` one, but with the capacity to distinguishes regions where there are mutations between the reference and target-sequence(s). To execute it, you can declare your variable `descriptor_col` like this:

```
descriptor_col = { 'fmac' :  [6, 20, 22, 54, 94, 46][3] }
```

Now the log file will also inform you that mutations are being distinguished by means of the `True` Boolean in the `mutation` column:

[3] Remember that you can also use the `random` method here.

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	fmac	NaN	[6, 20...]	True	NaN	NaN

A typical HTML outcome using the descriptor `fmac` appears in Figure 5.5.1. See that mutations are grayed out. If you want to change the Gray color of the mutations (default), you can do it through the `-mco` command that will be described in Section 5.13.

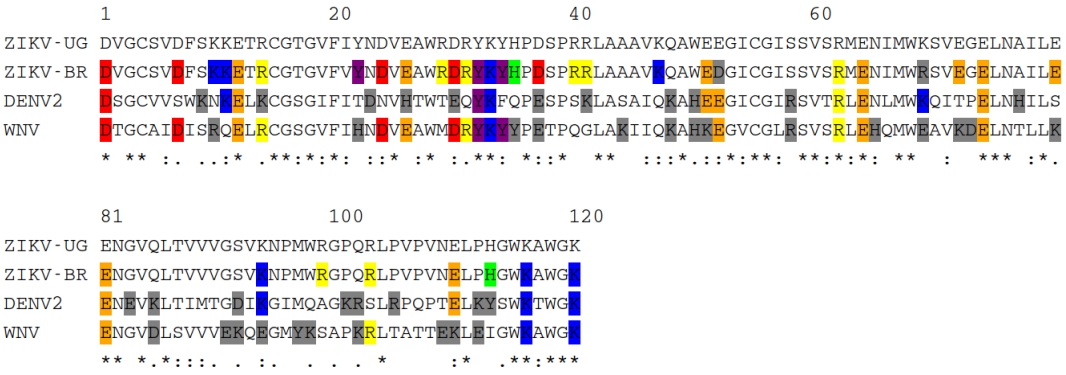


Figure 5.5.1: Alignment colored through `fmac` descriptor.

5.6 Color by position index (`pic`)

When we want to highlight only specific regions of the alignment we can use the `pic` descriptor, like this[4]:

```
descriptor_col = { 'pic' :  [ { '1' :  [6, 'n', '1-2', '3', '4-8'] } ,
                              { '1' :  [94, 'b', '11-27'] } ,
                              { '2' :  [75, 'd', '3', '30-55'] } ,
                              { '3' :  [54, 'u', '20-35'] } ] }
```

This can be one of the most complex color descriptors to implement. The descriptor `pic` is a dictionary type where its key will be the string `'pic'`, and the value is a list of dictionaries with several parameters (the orange square brackets delimit the list and purple brackets the dictionaries).

[4]Some brackets, letters, and numbers are colored to aid the reader.

We have seen this arrangement structure before (see Subsection 3.10.3). However, when `pic` is implemented we have some differences. Our second-key will be a number (string type), from one of the target-sequence(s) (for our example, `'1'`, `'2'`, and `'3'`, or ZIKV-BR, DENV2, and WNV, respectively).

Note: It is imperative that you respect the order in which the target-sequences appear in a specific comparison. However, note that you can call the same sequence more than once as `'1'` *in our example.*

The second-value is a list of parameters where the first (in red letters) and second (in green letters) refer, respectively, to the positive integer color number (see Section 5.16) and the font style `['n'` ("none"), `'b'` ("**bold**"), `'i'` ("*italic*"), `'d'` ("~~deleted~~"), and `'u'` ("underlined")]. From the third (in blue), we find the intervals (string type) that we want to color with the format: `'number`$_1$`-number`$_2$`'`. The intervals must be from a lower to a higher number and sometimes only one can be considered (for our example see `'3'` in blue). You should respect this consistency! The numbers of the intervals that you must implement are the same when the input data is processed (see the index of each row in Figure 3.2.1 or the "Data processed" section of the log file). Remember, your intervals should be related to that index and not the alignment position!

Note: Be careful about overlaps that can occur between the intervals, watch out for that.

Running CPRISMA we find the following information in the log file:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	pic	[1]	[6...]	Nan	n	NaN
main_0	1	pic	[1]	[94...]	Nan	b	NaN
main_0	1	pic	[2]	[75...]	Nan	d	NaN
main_0	1	pic	[3]	[54...]	Nan	u	NaN

Notice that for `pic` descriptor, the `sequence` column is not showing the color as we saw with the `fsc` descriptor, instead it is displaying the number of the target protein that is being colored. To find the color, you must see the first element in the list that appears in the `residue` column [for our example the color values will be `6` (Red), `94` (Blue), `75` (Cyan), and `54` (Lime)]. The next elements on the same list will be the highlighted residues[5]. Also the font letter that is being applied for each region will appear in the `style` column. The HTML file is going to display specific regions based

[5] In the table they are being replaced by points.

on the position index of the processed data (Figure 5.6.1). Make correlations with both that graph and the table above. Note that if the interval match with a gap, this region not be colored. For instance, the interval `'1-2'` (sequence$_1$/ZIKV-BR) presents a gap at position 2 (Figure 3.2.1), as this position coincides with a Lys residue for DENV2 and one Ser residue for ZIKV-BR, no color should be assigned to that region of ZIKV-BR, due to the Ser is not a target-residue.

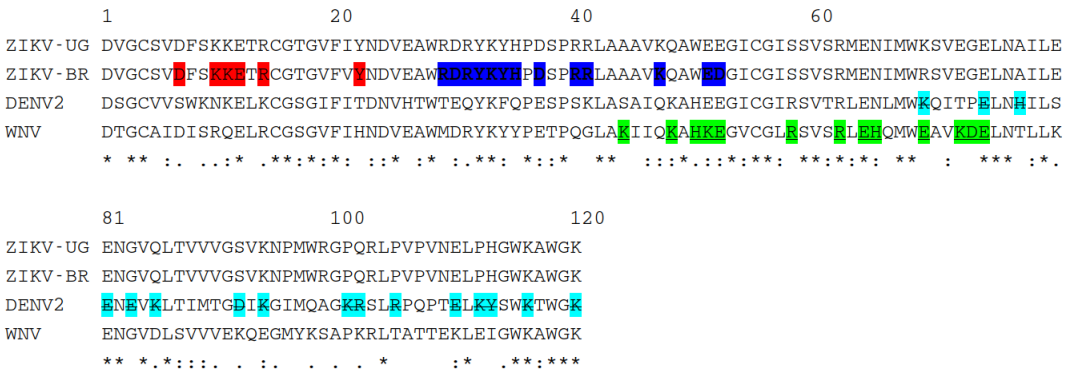


Figure 5.6.1: Alignment colored through `pic` descriptor.

5.7 Color by position index and mutation (`pimc`)

If you want to put color based on an index, but at the same time you want to distinguish point mutations you can implement `pimc`[6]. This descriptor is a copy of `pic`, so we can simply change `pic` by `pimc` from the previous array to get the following information:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	pimc	[1]	[6...]	True	n	NaN
main_0	1	pimc	[1]	[94...]	True	b	NaN
main_0	1	pimc	[2]	[75...]	True	d	NaN
main_0	1	pimc	[3]	[54...]	True	u	NaN

Check that the `mutation` column appears with the Boolean value equal to `True`. If you prefer another color for the mutations remember that it is also possible to implement the `-mco` flag (see Section 5.13).

[6]Imagine that this descriptor is equivalent to `fmac` or `tmc` (see Sections 5.5 or 5.9, respectively).

The HTML file will be like shown in Figure 5.7.1.

A practical example using this descriptor appears in Section 8.3. Also, the implementation of the `pimc` descriptor appears in a previously published study [3].

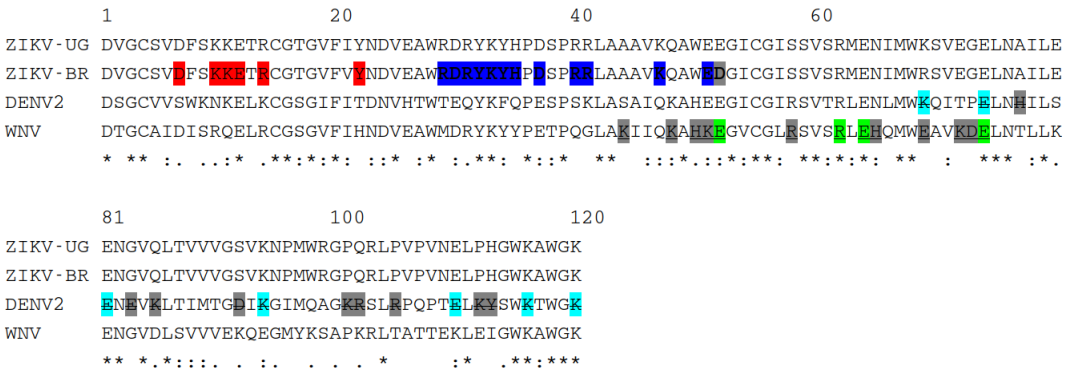


Figure 5.7.1: Alignment colored through `pimc` descriptor.

5.8 Color by threshold (`tc`)

When you want to discriminate the numeric values of the input data by means of color and using a threshold, it can be done through the following declaration:

```
descriptor_col = { 'tc' :  [94, 0.1, '>='] }
```

The descriptor `tc` is a dictionary where its value is a list with three elements. In the first position of that list, you must put a positive integer number that is related to the type of color desired (see Section 5.16). In second position, there is an `int`/`float` number that defines the threshold, and the third element, must be a string that refers to a symbol of order relation (*i.e.,* `'>='`, `'<='`, `'>'`, `'<'`, `'=='`, `'!='`). For our example we are ordering the program to color in Blue only those target-residues that are greater equal than 0.1.

If you run CPRISMA program, the log file will report the color number in the `sequence` column, and the two last parameters of the `tc` descriptor list in the `threshold` column, like this:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	tc	[94]	NaN	NaN	NaN	0.1, >=

The HTML file will now display a color for each target-residue following the threshold condition (Figure 5.8.1).

A practical example using this descriptor appears in Section 8.2. The implementation of the `tc` descriptor appears in a previously published study [2].

Figure 5.8.1: Alignment colored through `tc` descriptor.

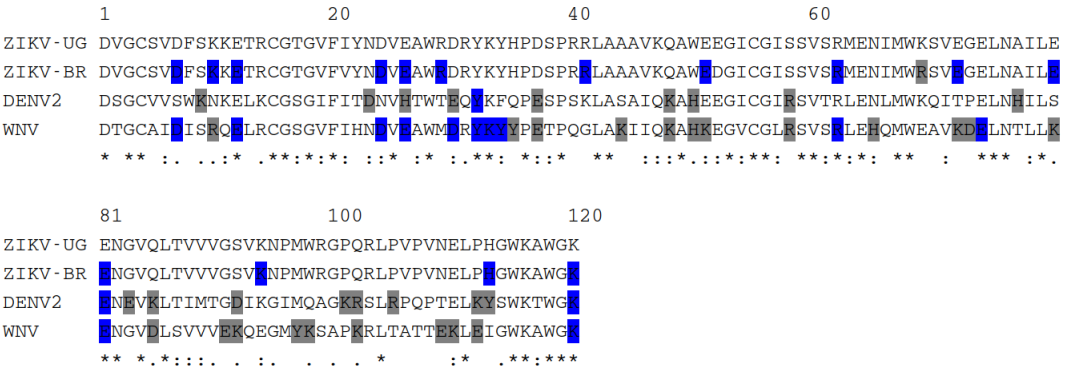


Figure 5.8.2: Alignment colored through `tmc` descriptor.

5.9 Color by threshold and mutation (`tmc`)

The descriptor `tmc` is the version of `tc` that will simultaneously discriminate mutations. This follows the same `tc` rules and is invoked as follows:

```
descriptor_col = { 'tmc' :  [94, 0.1, '>='] }
```

The log file will report the nex information:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	tc	[94]	NaN	True	NaN	0.1, >=

Look at the `mutation` column appears with the Boolean value equal to `True`. If you prefer another color for the mutations remember that it is also possible to implement the `-mco` flag (see Section 5.13).

The HTML file will be like shown in Figure 5.8.2. Note that CPRISMA only considers mutations that are matching the declared threshold condition.

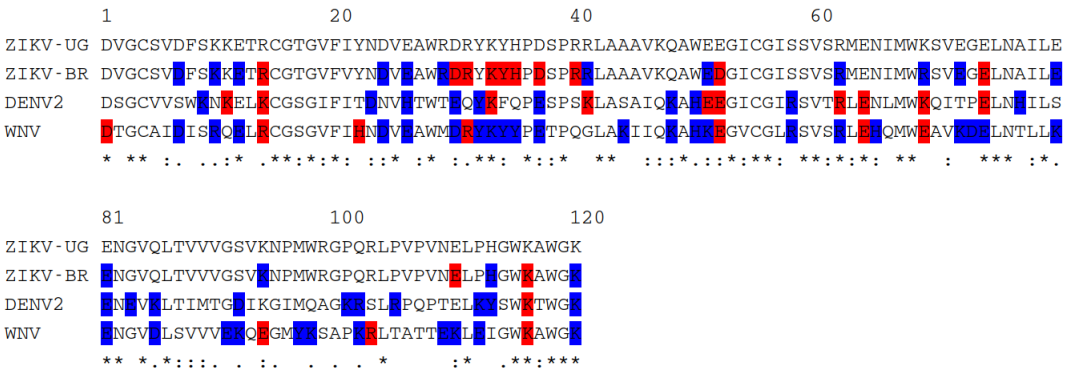


Figure 5.9.1: Alignment colored through `pkac` descriptor and invoking `False` as value.

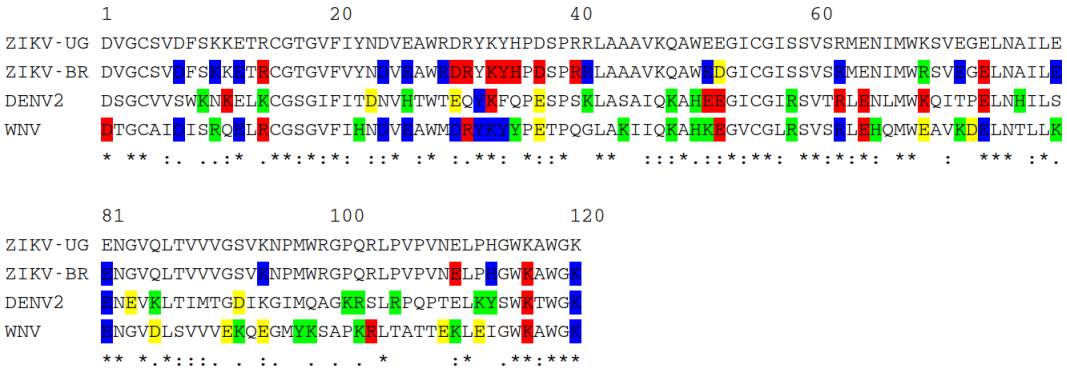


Figure 5.9.2: Alignment colored through `pkac` descriptor and invoking `True` as value.

5.10 pK_a color (pkac)

To use the `pkac` descriptor you must *always* do a Δ operation among *ionizable residues* (see Section 4.2). To execute this descriptor, your "array_get.py" script should contain the following instructions:

```
descriptor_col = { 'pkac' :  True/False[7] }
```

Note: It is imperative you declared the `descriptor_ope` *to be equal to* `{ 'd' }` *to get the expected results!*

The descriptor `pkac` is able to distinguish between those negative and positive ΔpK_a of the ionizable groups, and will assign Red and Blue colors, respectively. Observe that the `descriptor_col` dictionary has Boolean variable as value. You can interpret `False` or `True` for cases where you do not require (as `fac`/`pic`/`tc` descriptors) or require (as `fmac`/`pimc`/`tmc` descriptors) to distinguish point mutations, respectively. If `True`, mutations will be distinguished in two ways:

1. When there is a shift from acid residue to basic one (Lime color will be assigned).
2. Vice versa (Yellow color will be assigned).

A typical output in the log file when the `pkac` descriptor is invoked with `False` Boolean appears below:

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	pkac	NaN	[ionizable]	NaN	NaN	NaN

Note: Remember, if you invoke the Boolean `True` *as a value in the dictionary of* `pkac` *descriptor, then this parameter will be displayed on the* `mutation` *column as* `True`.

Whenever `pkac` descriptor is run, the `residue` column will show the information: `[ionizable]`. Examples of colored alignments by means of `pkac` descriptor using `False` and `True` appear in Figures 5.9.1 and 5.9.2, respectively. If you perceive, this descriptor can be quite useful for data with negative or positive numbers even if they are not pK_a values!

A practical example appears in Section 8.1. However, note that for these cases other features are being incorporated, such as color gradient (see Chapter 6). The implementation of the `pkac` descriptor appears in a previously published study [1].

[7] Select only one Boolean option.

5.11 Color descriptors dictionary (-dco)

When it is executed the `multiple` comparison method, it is possible that you need to distinguish each reference/target(s) relationship with a specific color descriptor. For that, we must build an array of color descriptors through the `dict_col` variable of the script "array_get.py". To build it, you should follow the same rules like for `dict_ope` (see Section 4.5). For this example, we used the same dictionary for the `multiple` method as a base (see Subsection 3.10.3). Then, our CPRISMA color-array will be as follows[8]:

```
dict_col = { 'main_0' :  [ { 'pimc' :  [ { '1' :  [63, 'b', '0-20'] } ,
                             { '3' :  [96, 'u', '20-35'] } ] } , { 'ssc' } ] ,
           'main_1' :  [ { 'fsc' :  [2,80] } , { 'pkac' :  True } ] ,
           'main_2' :  [ { 'fac' :  ['6, 20, 22, 54, 94, 46'] } ] }
```

Now we can execute the `-dco` flag to call the array `dict_col`:[9]

```
cprisma -tr -ck -rf multiple -va 2 -hc -sco 20 -mco 75 -dco
```

Your log file will show the relationships between each comparison and color descriptor:

```
Comparing dict_ref with dict_col:
                                {'0':  [1, 2, 3]} ........  { 'pimc' :  ...  }
                                {'0':  []} ........  { 'ssc' }
                                {'1':  [0, 2]} ........  { 'fsc' :  [2,80] }
                                {'1':  [0, 2, 3]} ........  { 'pkac' :  True }
                                {'2':  [0]} ........  { 'fac' :  [6, 20, 22...]  }
The array of the feature 'color' dict_col is compatible with the array of
comparison sequences dict_ref!
```

... and a table with a summary for all color descriptor information per comparison:

[8] Note that we have differentiated each color descriptor for each comparison with a different color to aid the reader.
[9] Notice that additional parameters are being implemented.

main	comparison	descriptor	sequence	residue	mutation	style	threshold
main_0	1	pimc	[1]	[63...]	True	b	NaN
main_0	1	pimc	[3]	[96...]	True	u	NaN
main_0	2	ssc	[same]	NaN	NaN	NaN	NaN
main_1	3	fsc	[2, 80]	NaN	NaN	NaN	NaN
main_1	4	pkac	NaN	[ionizable]	True	NaN	NaN
main_2	5	fac	NaN	[6...]	NaN	NaN	NaN

Figure 5.11.1 is showing the alignment for the typed commands and the built `dict_col` array.

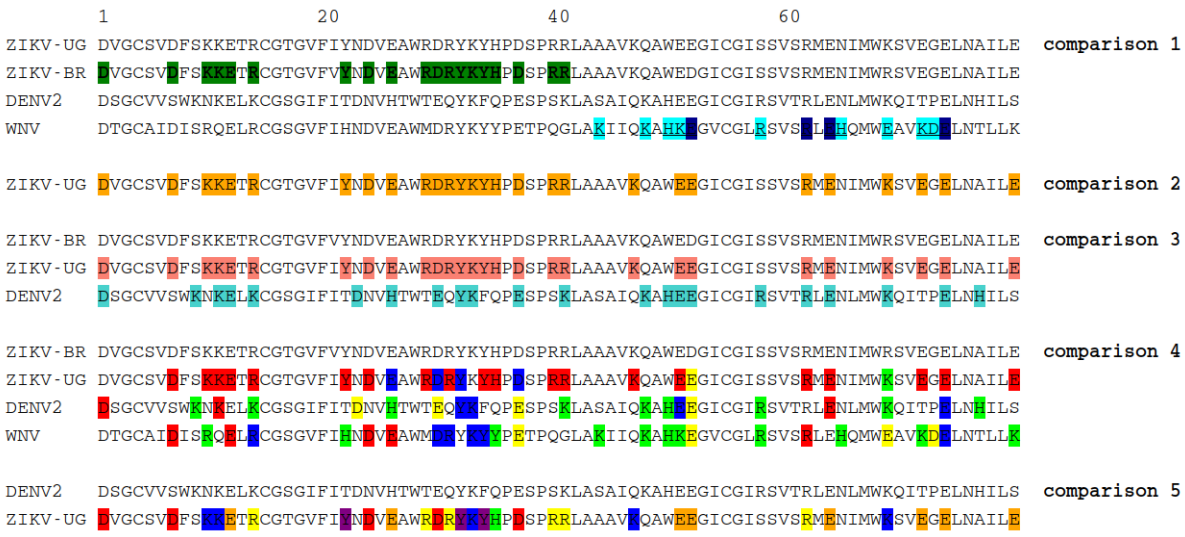


Figure 5.11.1: Alignment for `multiple` comparison method and colored through an array of color descriptors `dict_col`. Not all alignment is being shown.

Note: If the array of `dict_col` *is wrongly built, all descriptors will be transformed to* `nc` *(default parameter) for each comparison and additional information about where the errors are will be displayed.*

Note: Remember that these complex color-arrays can be simultaneously combined (or not) with other transcendental CPRISMA arrangements of the other features as well (i.e., operation, visualization, and maximum restriction described in Sections 4.5, 6.1, *and* 7.14*). See also the practical examples in* Chapter 8.

5.12 Color sequence (-sco)

To change the default color of the **ssc** descriptor, you should leave your variable **descriptor_col** as declared above in Section 5.2 and execute the **-sco** flag followed by positive integer number in the terminal, as follows:

```
cprisma -tr -ck -sco 51
```

Remember that this number should be equivalent to a positive integer number less equal than 142. For the command above we have changed the alignment color from Red to GreenYellow, which is at row-position 51 when **-lco** is executed (see Section 5.16).

5.13 Color mutation (-mco)

You can change the color of the mutations by executing a command like the one below:

```
cprisma -tr -ck -mco 75
```

Remember to use the same rules as for **-sco** flag (see Section 5.12). For the command above, it was changed the color of point mutations from Gray to Cyan.

Note: The flag **-mco** *only works for* **fmac**, **pimc**, *and* **tmc** *color descriptors (see* Sections 5.5, 5.7, *and* 5.9*).*

5.14 Color on three-dimensional structure (-tco)

If you want to observe the regions with color at the three-dimensional structure level, the **-tco** flag can be useful. Using the case of Section 5.10 as an example[10], we can execute:

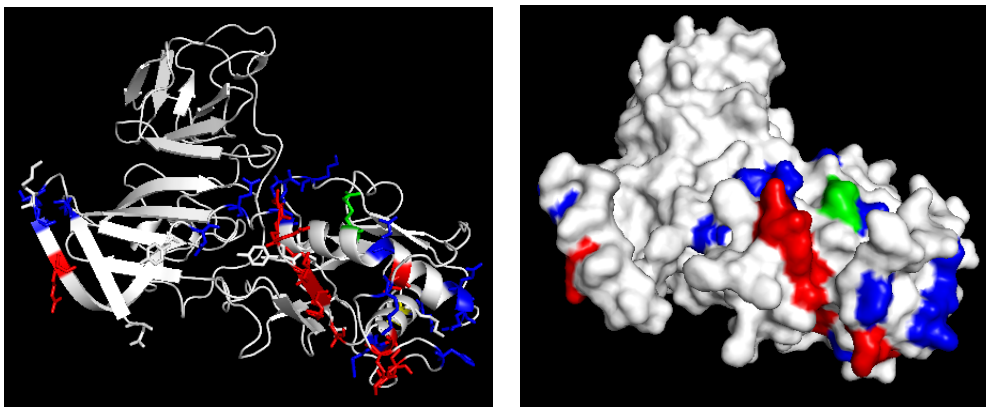
```
cprisma -tr -ck -tco
```

This command will generate a directory called "3D_representation" with various Pymol scripts for all proteins and comparisons. To visualize the protein, you can type:

```
pymol your-protein.pdb 3D_representation/main_0/main_0_comparison1_your-protein.pml
```

[10] Differences between mutations are being considered.

An example for ZIKV-BR appears in Figure 5.14.1. Notice that only the target-residues are taken into account. Furthermore, only the first 120 amino acids are being considered.



(a) (b)

Figure 5.14.1: Three-dimensional structure for the monomeric $NS1_{ZIKV\text{-}BR}$ (PDB id 5GS6) using the `pkac` descriptor. (a) Ribbons representation (target-residues appear as sticks). (b) Surface representation. Visualizations were made from Pymol scripts generated through the `-tco` flag.

5.15 Color intensity (`-ico`)

Sometimes when we apply a gradient color (see Chapter 6) the intensity of this can not be the best due to various problems (as mentioned earlier in the Section 3.2). An alternative to fix these dilemmas, it is to multiply the intensity of the color by a positive integer number invoking the `-ico` flag. As an example, you can execute it as appears below:

```
cprisma -tr -ck -ico 4
```

For the previous command line, we are multiplying by 4 the intensity of the color. We recommend the user to test with gradient colors and implement `-ico`.

5.16 List of colors (`-lco`)

To have quick access to the list of colors you can execute the command:

```
cprisma -lco
```

Figure 5.16.1 shows the first 10 rows displayed in the terminal as an example. See that an index appears numbering each color. This index will be useful to access the colors that CPRISMA apply. Remember, these colors are based on the website https://htmlcolorcodes.com/color-names/.

Note: The possible number of colors is from 0 to 142.

	General	Name	Hex	RGB
0	Red HTML	IndianRed	#CD5C5C	rgb(205, 92, 92
1	Red HTML	LightCoral	#F08080	rgb(240, 128, 128
2	Red HTML	Salmon	#FA8072	rgb(250, 128, 114
3	Red HTML	DarkSalmon	#E9967A	rgb(233, 150, 122
4	Red HTML	LightSalmon	#FFA07A	rgb(255, 160, 122
5	Red HTML	Crimson	#DC143C	rgb(220, 20, 60
6	Red HTML	Red	#FF0000	rgb(255, 0, 0
7	Red HTML	FireBrick	#B22222	rgb(178, 34, 34
8	Red HTML	DarkRed	#8B0000	rgb(139, 0, 0
9	Pink HTML	Pink	#FFC0CB	rgb(255, 192, 203
10	Pink HTML	LightPink	#FFB6C1	rgb(255, 182, 193

Figure 5.16.1: First 10 rows when invoked `-lco` command.

Chapter 6

Visualization (`descriptor_vis`)

Previously, in Chapters 3 and 5, we already saw some commands and variables that alter the display of alignments. In this Chapter, we will give an expansion focusing on three possible functionalities available on CPRISMA through the so-called "visualization" feature:

1. Reference display (`Re`).
2. Target-sequence(s) degrade (`De`).
3. Visualization of the letters (`Le`).

These three variables will be the descriptors of the "visualization" feature, and they are always declared simultaneously through the `descriptor_vis` of the "array_get.py" script (Figure 2.2.3), as follows:

```
descriptor_vis = { 'ReY', 'DeN', 'LeY' }
```

As you can see, the `descriptor_vis` is a set variable with three elements. In addition, each descriptor is accompanied by a letter `Y` (Yes) or `N` (No), which indicates whether or not this visualization feature is enabled, respectively.

Suppose we declare the operation (see Chapter 4) and color (see Chapter 5) variables like this,

```
descriptor_ope = { 'd' }
```

```
descriptor_col = { 'pkac' :  True }
```

... and we run the next command[1]:

```
cprisma -tr -ck
```

You may have noticed that your log file shows the following information:

main	comparison	reference	degraded	letters
main_0	1	True	False	True

As you can see for each visualization descriptor, we will have a Boolean variable that is related to `Y` (`True`) and `N` (`False`).

Note: Sometimes, it could be declared some descriptors wrongly. To avoid that the program stops, it will always return to `'ReY', 'DeN', 'LeY'` *descriptors (default parameters), with some additional error messages in your log file, which will guide you to correct the problem.*

Early, we saw the output aligment for these initial parameters (Figure 5.9.2). But now, let us change `ReY` by `ReN`, as follows:

```
descriptor_vis = { 'ReN', 'DeN', 'LeY' }
```

As you can see in Figure 6.0.1a, the "reference line" has disappeared. Since this sequence may not be relevant because it does not have any color, the `Re` descriptor allows us to remove it or not as appropriate.

Enabling the degrade, like this:

```
descriptor_vis = { 'ReY', 'DeY', 'LeY' }
```

... we can see the effect that our operations have when we invoked `d` on the `descriptor_ope` (Figure 6.0.1b). Check that the color is on a scale from -13.3 to 13.3. This scale is being normalized based on the maximum value derived from Δ operation[2]. We observe that the point mutations have a greater impact on the rest of the $\Delta \mathrm{p}K_a$ values. Nevertheless, it is possible to apply maximum restriction methods to improve the color alignment display (see Chapter 7).

Finally, if we are not interested in visualizing the letters of the protein sequences, you could eliminate them by means of the following declaration:

[1] It is important to mention that for most of the examples in this Chapter we have used this command and these variables as declared.

[2] See explicitly the Δ calculations in your log file with the `-t` flag (see Section 3.11).

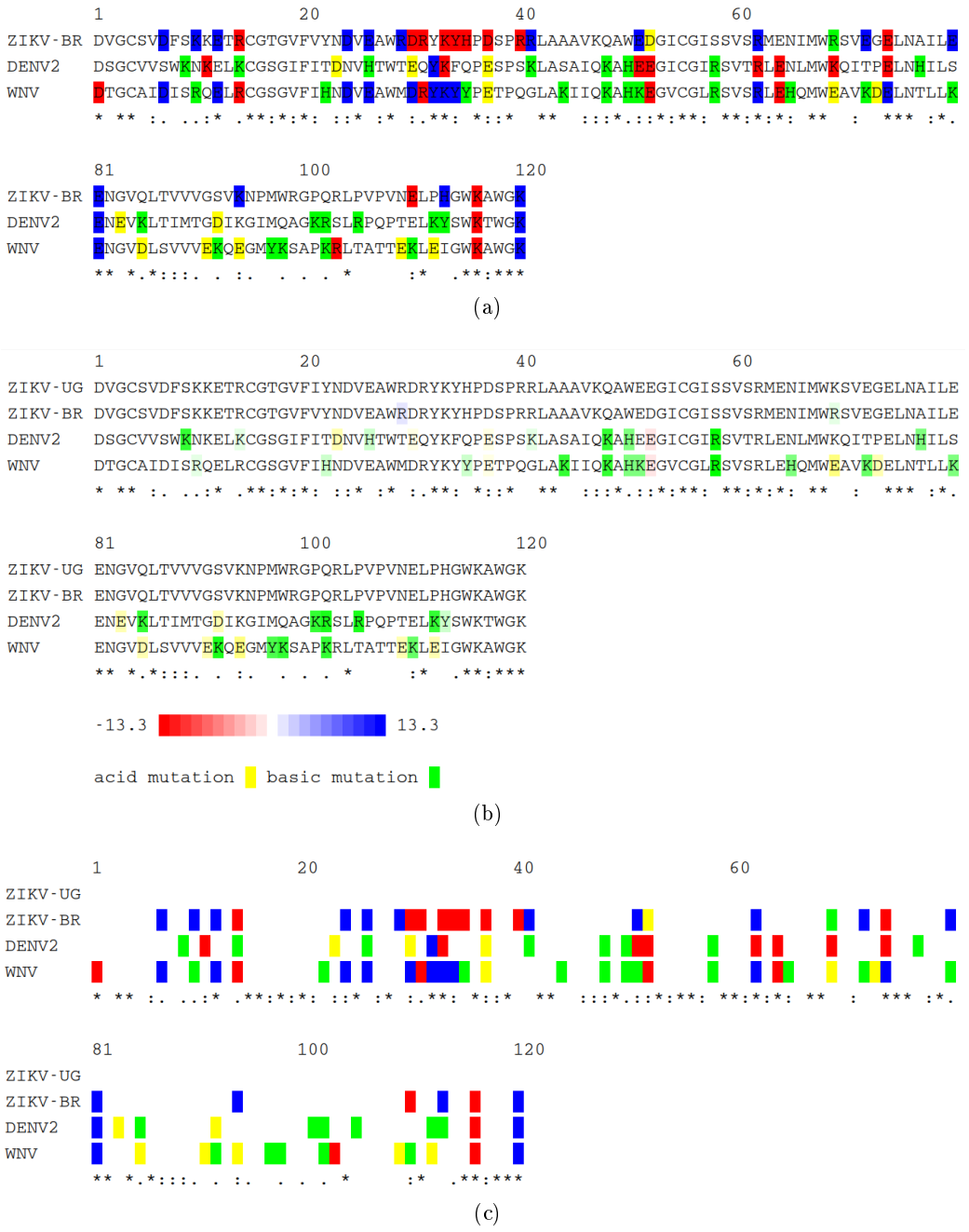


Figure 6.0.1: Several alignments using different visualization descriptors. (a) `Re` descriptor with `False` Boolean (`N`). (b) `De` descriptor with `True` Boolean (`Y`). (c) `Le` descriptor with `False` Boolean (`N`).

```
descriptor_vis = { 'ReY', 'DeN', 'LeN' }
```

Figure 6.0.1c shows an alignment without the letters.

We recommend to check the practical examples in Sections 8.1 and 8.2 for a better understanding of the visualization descriptors.

6.1 Visualization descriptors dictionary (`-dvi`)

When you make trascendental comparisons with the `multiple` method (see Subsection 3.10.3), perhaps you will require to apply a specific visualization based on `Re`, `De`, and `Le`, for each reference/target(s) relationship. For this, we can invoke the `-dvi` flag to get a visualization descriptors dictionary `dict_vis` from the "array_get.py" script. It is important to mention that we will follow the same rules as already described for `dict_ope` (see Section 4.5). For our example, we will simultaneously apply the `dict_col` arrangement mentioned in Section 5.11. On the other hand, it will be assumed `d` for all comparisons (*i.e.*, `descriptor_ope = { 'd' }`) and the same array of comparisons described in Subsection 3.10.3. Our `dict_vis` variable will be declared as follows[3]:

```
dict_vis = { 'main_0' :  [ { 'ReN', 'DeN', 'LeY' } , { 'ReY', 'DeY', 'LeY' } ] ,
             'main_1' :  [ { 'ReY', 'DeN', 'LeN' } , { 'ReY', 'DeY', 'LeY' } ] ,
             'main_2' :  [ { 'ReN', 'DeY', 'LeY' } ] }
```

Now we can execute the `-dvi` flag to call the array `dict_vis`[4]:

```
cprisma -tr -ck -rf multiple -va 2 -hc -sco 20 -mco 75 -dco -dvi
```

Your log file will show the relationships between each comparison and visualization descriptors:

```
Comparing dict_ref with dict_vis:
                                {'0':  [1, 2, 3]} ........  ['ReN', 'DeN', 'LeY']
                                {'0':  []} ........  ['ReY', 'DeY', 'LeY']
                                {'1':  [0, 2]} ........  ['ReY', 'DeN', 'LeN']
```

[3] Note that we have differentiated each set of visualization descriptors for each comparison with a different color to aid the reader.

[4] Notice that additional parameters are being implemented.

```
{'1':  [0, 2, 3]} ........  ['ReY', 'DeY', 'LeY']

{'2':  [0]} ........  ['ReN', 'DeY', 'LeY']
```

`The array of the feature 'visualization' dict_vis is compatible with the array of comparison sequences dict_ref!`

... and a table with a summary for all visualization descriptor information per comparison:

main	comparison	reference	degraded	letters
main_0	1	False	False	True
main_0	2	True	True	True
main_1	3	True	False	False
main_1	4	True	True	True
main_2	5	False	True	True

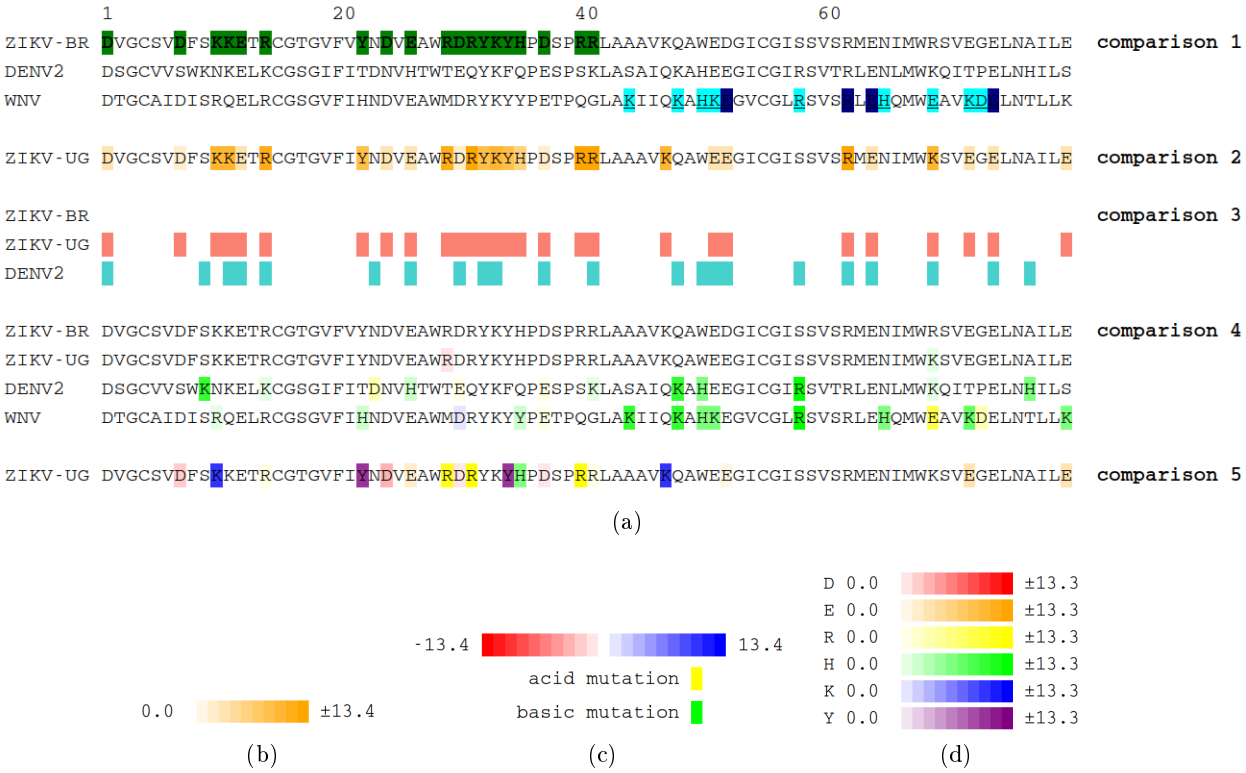


Figure 6.1.1: (a) Alignment for `multiple` comparison method and colored through the arrays of color (`dict_col`) and visualization (`dict_vis`) descriptors. Not all alignment is being shown. (b) Scale for comparison 2. (c) Scale for comparison 4. (d) Scale for comparison 5.

The HTML output for the previous conditions invoked is shown in Figure 6.1.1a. Make the respective relationships between the previous table and that graph. Since we are invoking `DeY` for comparisons `2`, `4`, and `5`, we are obtaining, respectively, their scales in the log file (Figures 6.1.1b, 6.1.1c, and 6.1.1d). Despite comparison `2` does not have any target-sequence (*i.e.*, `{'0':[]}`), the gradient color that you are seeing for this case is related to reference protein. The case of comparison `4` is similar to what we saw earlier (Figure 6.0.1b), only now the reference sequence is ZIKV-BR. Finally, see that comparison `5` has separated a different color scale for each target-residue. This is because the color descriptor `fac` is being declared for that comparison (see Section 5.4). However, the scale is normalized based on the greater value of the result of the operation `d` that is being applied for that comparison (*i.e.*, ±13.3). You can see the independent behavior for each amino acid, but for that, you need to apply special maximum restriction methods (see Section 7.13).

Note: If the array of `dict_vis` *is wrongly built, all descriptors will be transformed to* `ReY, DeN, LeY` *(default parameters) for each comparison and additional information about where the errors are will be displayed.*

Chapter 7

Maximum restriction (descriptor_mxr)

To understand how the CPRISMA feature called "maximum restriction" works, let's suppose that we declare in the "array_get.py" script the following variables like this[1]:

```
descriptor_ope = { 'd' }

descriptor_col = { 'pkac' :  True }

descriptor_vis = { 'ReY', 'DeY', 'LeY' }
```

Then the following command is run in terminal[2]:

```
cprisma -tr -ck -t
```

The HTML output for this example was already cited earlier (Figure 6.0.1b). Although this output is correct, we are not clearly observing the pK_a shifts for many target-residues. Since we have invoked the `-t` flag (see Section 3.11), our log file will show in the operation feature part, how the Δ data is being obtained (see Figure 7.0.1 as an example). Check that the value of ΔpK_a highlighted in row 13 (*i.e.*, -13.3) is based on the subtraction p$K_{a-\text{DENV2}}$ $-$ p$K_{a-\text{ZIKV-UG}}$. This value match with a mutation, due to a genetic indel (see row 13 in Figure 3.2.1). Thus, the scale of Figure 6.0.1b is being normalized as a function of this value, which corresponds to the maximum among the entire data set. However, the regions that should be more Red or Blue cannot display that color because their ΔpK_a are very low with respect to the greater value. For instance, taking $\left|0.1^{\Delta \text{p}K_a} / -13.3^{\Delta pK_a(\max)}\right|$, we

[1] For a further explanation of these variables see Chapters 4, 5, and 6, respectively.
[2] Most of the examples below are using the this command line.

see that 0.1 is 7.5×10^{-3} smaller than the maximum value. From the point of view of the HTML code, this is imperceptible and the color should be equal to White.

To improve the color displayed when is invoked a gradient by `DeY` (see Chapter 6), we can use a maximum restriction method like: `rm`, `ra`, `ram`, `rpi`, `rt`, `rs`, `rsm`, `rsa`, `rsam`, `rspi`, and `rspim`, through the variable `descriptor_mxr` of the script "array_get.py" (Figure 2.2.3).

ZIKV-UG X ZIKV-BR DENV2 WNV

	0	1	2	3	d			
0	3.9	3.9	3.9	3.8	->	0.0	0.0	-0.1
1	3.1	3.4	0.0	3.4	->	0.3	-3.1	0.3
2	0.0	0.0	10.5	0.0	->	0.0	10.5	0.0
3	10.9	11.0	0.0	12.8	->	0.1	-10.9	1.9
4	10.7	10.7	10.6	0.0	->	0.0	-0.1	-10.7
5	3.8	4.0	3.8	4.0	->	0.2	0.0	0.2
6	12.6	12.3	10.7	12.4	->	-0.3	-1.9	-0.2
7	10.0	10.0	0.0	6.7	->	0.0	-10.0	-3.3
8	0.0	0.0	3.4	0.0	->	0.0	3.4	0.0
9	3.5	3.8	0.0	3.6	->	0.3	-3.5	0.1
10	4.0	4.1	6.8	4.4	->	0.1	2.8	0.4
11	12.6	13.4	0.0	0.0	->	0.8	-12.6	-12.6
12	3.1	2.7	4.1	3.5	->	-0.4	1.0	0.4
13	13.3	13.0	0.0	12.9	->	-0.3	-13.3	-0.4
14	9.6	9.6	10.0	9.8	->	0.0	0.4	0.2
15	11.1	11.0	11.0	11.2	->	-0.1	-0.1	0.1
16	10.0	9.7	0.0	10.2	->	-0.3	-10.0	0.2
17	6.3	6.2	0.0	9.5	->	-0.1	-6.3	3.2

Figure 7.0.1: Example of the pK_a data for each target protein and its ΔpK_a obtained after running `d` as operation. The number highlighted in blue refers to the maximum value for this data set. All the information has not been showed.

7.1 No restriction (nr)

When it is not considered maximum restrictions we declared the `descriptor_mxr` as follows:

```
descriptor_mxr = { 'nr' }
```

You can perceive that your `nr` descriptor is a string variable inside a pair of brackets. It is worth to mention that for all the previous examples this was the maximum restriction condition implemented.

When we run CPRISMA, our log file will show the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	nr	NaN	NaN	NaN	NaN	NaN	NaN

This statement is simply saying that `nr` descriptor was considered for comparison 1, in the `main_0` comparison set (for now ignore the rest of the columns in that log message). Besides this table, in the maximum restriction part of the log file, you will find a list with all values for the operations invoked, like this[3]:

```
list_max = [0.0, 0.3, 0.0, 0.1, 0.0, 0.2, -0.3, 0.0, 0.0, 0.3, 0.1, 0.8, -0.4, -0.3, 0.0, -0.1,
-0.3, -0.1, -0.5, -0.2, 0.1, 0.0, 0.0, 0.0, 0.0, 0.1, -0.6, 0.0, 0.1, 0.0, 0.0, 1.7, 0.2, 0.0,
-0.1, 0.0, 0.1, 0.3, 0.0, 0.0, 0.0, 0.0, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.2, 0.0,
0.1, -0.5, 0.3, 0.0, -3.1, 10.5, -10.9, -0.1, 0.0, -1.9, -10.0, 3.4, -3.5, 2.8, -12.6, 1.0, -13.3,
0.4, -0.1, -10.0, -6.3, 0.7, -13.0, -1.8, 0.0, -11.0, 10.9, 6.4, -0.3, -0.8, 12.9, -0.2, -0.1, 0.0,
-0.2, -3.8, 0.0, -0.2, 6.8, -4.2, 0.1, 4.2, 10.8, 0.0, 3.5, 0.0, 0.0, 0.0, -12.7, 10.2, 12.4,
-12.9, 12.6, 0.0, 0.0, 11.2, 3.3, -0.2, 0.1, -0.1, 0.3, 0.0, 1.9, -10.7, 0.2, -0.2, -3.3, 0.0, 0.1,
0.4, -12.6, 0.4, -0.4, 0.2, 0.1, 0.2, 3.2, 0.7, -13.0, -12.9, 10.7, -11.0, 10.6, 6.1, 6.5, -0.9,
13.2, 0.1, -0.2, 6.8, -7.3, 7.9, 3.4, 0.1, 0.0, 7.2, 0.2, 0.0, 3.7, 4.0, 11.5, -6.7, 9.9, 10.9,
-12.7, 0.0, 10.2, -0.5, 0.0, 3.6, 7.6, 4.2, -6.4, -0.1, 0.2]
```

A summary of this information should appear immediately, as follows:

main	minimum (abs)	maximum (abs)	lenght
main_0	0.0	13.3	168

Notice that this table is showing the maximum and minimum numbers for the last list after applying the absolute value. This is because the HTML code will only work with positive numbers. Furthermore, the column `length` is showing us the total number of elements for that list ($56^{\text{rows}} \times 3^{\text{target-proteins}} = 168$). As no restriction applies, all data is being considered.

Note: Sometimes, it could be declared this descriptor or another wrongly. To avoid that the program stops, it will always return to `nr` *descriptor (default parameter), with some additional error messages in your log file, which will guide you to correct the problem.*

7.2 Restriction by mutation (`rm`)

For our example you may notice that most of the high ΔpK_a values correspond to regions where mutations occurred (compare Figures 3.2.1 and 7.0.1). To do not consider this "high ΔpK_a values", we can invoked the `rm` descriptor by means of:

```
descriptor_mxr = { 'rm' }
```

[3] These lists only appear when `-t` flag is executed (see Section 3.11).

Executing the program our table for this CPRISMA feature, it will appear in the log file like this:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rm	NaN	NaN	NaN	True	NaN	NaN

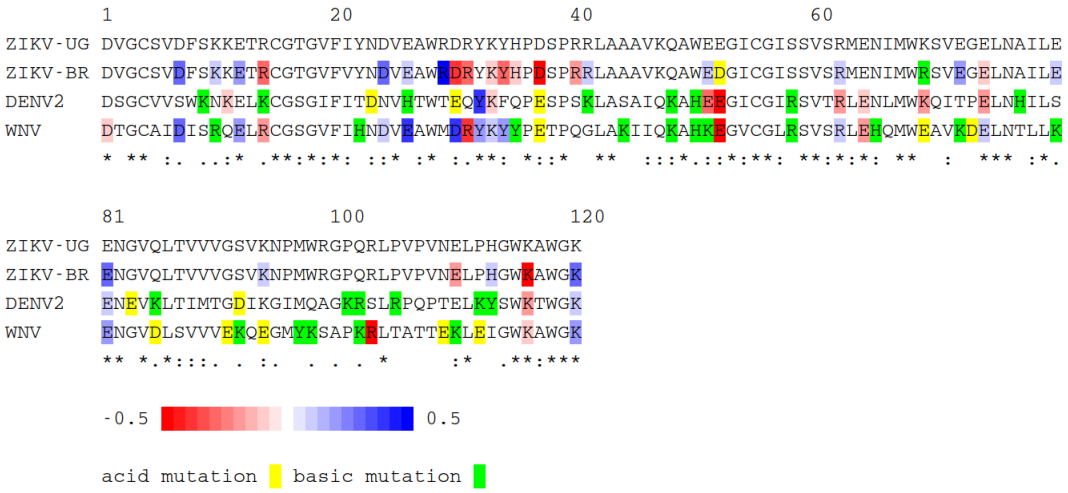


Figure 7.2.1: Alignment colored applying `rm` descriptor.

Notice that the `mutation` column has a Boolean equal to `True`. If we look at the lists of values that are being considered to normalize the color, we will find that our data is reduced:

```
list_max = [0.0, 0.2, 0.0, -0.1, 0.1, 0.0, -0.1, 0.3, -0.5, 0.3, 0.0, 0.0, 0.4, -0.1, -0.2, -0.1,
-0.2, 0.1, -0.2, 0.1, -0.1, 0.2, 0.2, 0.1, 0.1, -0.2, 0.1, 0.2, -0.1, 0.2]
```

main	minimum (abs)	maximum (abs)	lenght
main_0	0.0	0.5	30

Note: For the following sections and/or descriptors, this information will not be displayed or discussed anymore. When a maximum restriction is invoked, it is understood that CPRISMA is removing or transforming the data to normalize and improve the color visual quality on the alignment.

Changing the scale of these data allows us to observe with greater definition the color of where the pK_a shifts occur (Figure 7.2.1). At this point, we can finally see the impact of the numerical data of the CSV input file after Δ operation is applied.

7.3 Restriction by amino acid (ra)

Suppose you do not want to consider the numerical values of some target-residues (see Section 3.1). For that, you can invoke the `ra` descriptor, like this:

```
descriptor_mxr = { 'ra' :  ['R', 'K'] }
```

As we have already described in previous cases (see Chapter 5), the dictionary structure of the `ra` descriptor is compose by a key as a string (*i.e.*, `'ra'`) and a value as a list of target-residues (*i.e.*, `['R', 'K']`). For this example, CPRISMA is being instructed to do not take into account the numerical values of the row where Arg and Lys residues match in the processed data (for instance, row 2, 3, and 4 in Figure 3.2.1).

Note: When the `ra` *descriptor is run, CPRISMA will looks for all the rows where the residue declared appears and the program will "remove" the entire row if matchs. Later, CPRISMA will calculate the maximum from these transformations. If you prefer do not to delete the entire row, it is possible to use other more specific descriptors like* `rsa` *(see* Section 7.9*).*

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	ra	NaN	['R', 'K']	NaN	NaN	NaN	NaN

See that the `residue` column is showing the list declared. The color scale is now normalized based on this condition.

7.4 Restriction by amino acid and mutation (ram)

You can simultaneously constrain numerical values of target-residues and mutations like this:

```
descriptor_mxr = { 'ram' :  ['R', 'K'] }
```

The `ram` descriptor is a combination of `rm` (see Section 7.2) and `ra` (see Section 7.3), therefore it will follow the same rules as for those descriptors.

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	ram	NaN	['R', 'K']	NaN	True	NaN	NaN

Notice that the `mutation` column has a Boolean equal to `True`. Removing rows with Arg and Lys residues has changed the color scale. Now, it is displayed a maximum of ±0.4 instead of ±0.5 (Figures 7.4.1 and 7.2.1, respectively).

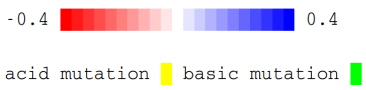


Figure 7.4.1: Scale example invoking `ram` descriptor.

7.5 Restriction by index position (`rpi`)

To have more control of the exact positions that you do not want to take into account you can declare the maximum restriction variable like this:

```
descriptor_mxr = { 'rpi' :  [13, 45, 48] }
```

The `rpi` descriptor has a dictionary structure similar to `ra` (see Section 7.3). However, the difference is that its value is a list of numbers that match with the row index of the processed data (Figures 3.2.1 and 7.0.1). Remember, the numbers of the list should be related to that index and not the alignment position!

Note: When the `rpi` *descriptor is run, CPRISMA will looks for all the rows based on the index positions declared and the program will "remove" the entire row if matchs. Later, CPRISMA will calculate the maximum from these transformations. If you prefer do not to delete the entire row, it is possible to use other more specific descriptors like* `rspi` *(see* Section 7.11*).*

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rpi	NaN	NaN	[13, 45, 48]	NaN	NaN	NaN

As you can see, in the `position` column, the list declared for the `rpi` descriptor appears. Based on this, the color scale will be normalized.

7.6 Restriction by threshold (`rt`)

If it is desired to apply maximum restrictions through a threshold, we can do it like this:

```
descriptor_mxr = { 'rt' :  [0.0, '>='] }
```

Regarding `rt` descriptor, we have a different dictionary compare to `ra` (see Section 7.3) or `rpi` (see Section 7.5). As you can see, the dictionary value is a list with two elements where the first shows a numeric value (float type) and the second a string that refers to a symbol of order relation (*i.e.*, `'>='`, `'<='`, `'>'`, `'<'`, `'=='`, `'!='`). Based on our example, we are telling the program to only take into account those rows with values greater equal than 0.0. If the row presents any number in any column that does not meet that condition, then, the value will be convert to 0.0 immediately.

Note: Occasionally, too strict a threshold can be invoked and this will cause all values to be ignored. To avoid the program stops, by default, CPRISMA will normalize everything based on 1.0.

The table when invoked `rt` descriptor looks like this:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rt	NaN	NaN	NaN	NaN	[0.0, >=]	NaN

If it is invoked `-t` flag (see Section 3.11), the log file will display the following information:

main	minimum (abs)	maximum (abs)	lenght
main_0	0.0	13.2	168

See that the `length` column is reporting the same amount of data as when we have no restriction (*i.e.*, when `nr` is applied [see Section 7.1)]. This is because all the values that do not satisfy the threshold condition are being converted to 0.0 and are not being "eliminated".

Finally, the new maximum for the color scale will be 13.2. Our maximum constraint has not changed much despite the convertion of the highest negative value (*i.e.*, from -13.3 to 0.0). As the threshold condition cited above does not take into account values less than 0.0, the problem persist because, even with this restriction, the data still have "high" positive values like 13.2.

Note: Based on the last statement, it is important to mention that `rt` *descriptor is more suitable for data with only one type of mathematical sign. For that you could use* `da` *as a operation descriptor (see* Section 4.3*).*

7.7 Restriction by sequence (rs)

In CPRISMA you can not consider the data of a whole sequence by invoking `descriptor_mxr` like this:

```
descriptor_mxr = { 'rs' :  [2] }
```

The dictionary structure of the `rs` descriptor is similar to `ra` (see Section 7.3) or `rpi` (see Section 7.5). However, in the list, you must place the target-sequence number to which you do not want to take into account the numerical values.

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rs	[2]	NaN	NaN	NaN	NaN	NaN

For our example, we are not taking into account the data of the DENV2 sequence. Based on this, the color scale will be normalized.

Note: Regarding the descriptor `rs`*, the* `pair` *comparison method, array of comparisons like* `{# : []}`*, or cases where* `-ck` *is not invoked, CPRISMA will always return to the default value of the maximum restriction feature (i.e.,* `nr`*).*

If you want to restrict the maximum values of target-residues or row position index from a specific sequence, we recommend Sections 7.9 and 7.11, respectively.

7.8 Restriction by sequence and mutation (rsm)

You can simultaneously "remove" the data from a sequence and the places where mutations appear by means of:

```
descriptor_mxr = { 'rsm' :  [2] }
```

The descriptor `rsm` is a combination of `rm` (see Section 7.2) and `rs` (see Section 7.7), so the same rules will apply for it.

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rsm	[2]	NaN	NaN	True	NaN	NaN

Notice that the `mutation` column has a Boolean equal to `True`. The scale of your HTML output will be normalized taking into account these parameters.

7.9 Restriction by sequence and amino acid (`rsa`)

Sometimes, it could be very radical to apply `ra` (see Section 7.3) due to many rows with useful values may be being removed. Nevertheless, to focus only on target-residues of a specific sequence, the `rsa` descriptor can be invoked. To declare this maximum restriction descriptor you can do it as follows:

```
descriptor_mxr = { 'rsa' : [ { '1' : ['R', 'K'] } , { '2' : ['D', 'Y'] } ] }
```

Notice that we saw this array structure before (see Subsection 3.10.3). Now, our value is a list of dictionaries. Each second-key represents the target-sequences of interest (*i.e.*, `'1'` and `'2'`) and the second-values are the lists of target-residues that we want to discard (*i.e.*, `['R', 'K']` and `['D', 'Y']`, respectively). For our example, CPRISMA is being instructed to just consider the sequences of ZIKV-BR (`'1'`) and DENV2 (`'2'`) and "to delete" the Arg and Lys residues; and the Asp and Tyr residues, respectively of that proteins. The descriptor `rsa` follows similar rules like `ra` but just considering the target-residues of the target-sequence of interest.

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rsa	[1]	['R', 'K']	NaN	NaN	NaN	NaN
main_0	1	rsa	[2]	['D', 'Y']	NaN	NaN	NaN	NaN

The color scale will be normalized based on these conditions.

Note: Regarding the descriptor `rsa`*, the* `pair` *comparison method, array of comparisons like* `{# : []}`*, or cases where* `-ck` *is not invoked, CPRISMA will always return to the default value of the maximum restriction feature (i.e.,* `nr`*).*

7.10 Restriction by sequence, amino acid, and mutation (rsam)

We can use a version of **rsa** that also restricts mutations simultaneously. For that, we could invoke the maximum restriction variable by means of:

```
descriptor_mxr = { 'rsam' :  [ { '1' :  ['R', 'K'] } , { '2' :  ['D', 'Y'] } ] }
```

The **rsam** is a combination of **rm** and **rsa** descriptors (see Sections 7.2 and 7.9, respectively), so it will follow the similar rules as them. For this case, **rsam** will not take into account the mutations that occur between the target-sequence of the second-key [for our example, '1' (ZIKV-BR) and '2' (DENV2)] and the reference sequence [for our example, '0' (ZIKV-UG)].

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rsa	[1]	['R', 'K']	NaN	True	NaN	NaN
main_0	1	rsa	[2]	['D', 'Y']	NaN	True	NaN	NaN

Notice that the **mutation** column has a Boolean equal to **True**. The color scale will be normalized based on these conditions.

7.11 Restriction by sequence and index position (rspi)

The **rspi** descriptor is similar to **rsa**, however the second-value must have the index positions of the processed data (see the index in Figure 7.0.1). To declare **rspi**, we can do it by means of:

```
descriptor_mxr = { 'rspi' :  [ { '1' :  [0, 29] } , { '2' :  [48] } ] }
```

The **rspi** descriptor shares the same rules as **rpi** (see Section 7.5) and **rsa** (see Section 7.9). For our example, CPRISMA is being instructed to just "delete" the positions 0 and 29 of ZIKV-BR ('1') and 48 of DENV2 ('2').

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rspi	[1]	NaN	[0, 29]	NaN	NaN	NaN
main_0	1	rspi	[2]	NaN	[48]	NaN	NaN	NaN

The color scale will be normalized based on these conditions.

Note: Regarding the descriptor `rspi`*, the* `pair` *comparison method, array of comparisons like* `{# : []}`*, or cases where* `-ck` *is not invoked, CPRISMA will always return to the default value of the maximum restriction feature (i.e.,* `nr`*).*

7.12 Restriction by sequence, index position, and mutation (`rspim`)

The version of `rspi` that restricts places where point mutations occur is called `rspim` and can be declared as follows:

```
descriptor_mxr = { 'rspim' :  [ { '1' :  [0, 29] } , { '2' :  [48] } ] }
```

The `rspim` is a combination of `rm` and `rspi` descriptors (see Sections 7.4 and 7.11, respectively), so it will follow similar rules as them[4].

The log file will display the following information:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rspim	[1]	NaN	[0, 29]	True	NaN	NaN
main_0	1	rspim	[2]	NaN	[48]	True	NaN	NaN

Notice that the `mutation` column has a Boolean equal to `True`. The scale of your HTML output will be normalized taking into account these parameters.

7.13 Restrictions by target-residues (`Y`)

As we can see, sometimes some target-residues have more influence than others when we apply a maximum restriction method. We can see the independent maximum behavior for each target-residue by adding the letter `'Y'` to our descriptor in the following way:

```
descriptor_mxr = { 'nrY' }
```

When we run CPRISMA, our log file will show the following information:

[4] See other considerations on how numerical data is removed at the places where mutations happen in Section 7.10 (although `rsam` and `rspim` are different maximum restriction methods they follow the same pattern of execution)

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	nr	NaN	NaN	NaN	NaN	NaN	True

Notice that the `separate` column is showing a Boolean variable as `True`. If it is invoked `-t` flag (see Section 3.11), the log file will display the following information:

main	minimum (abs)	maximum (abs)	lenght
main_0	0.0	[3.7, 7.3, 13.2, 6.8, 11.5, 9.9]	NaN

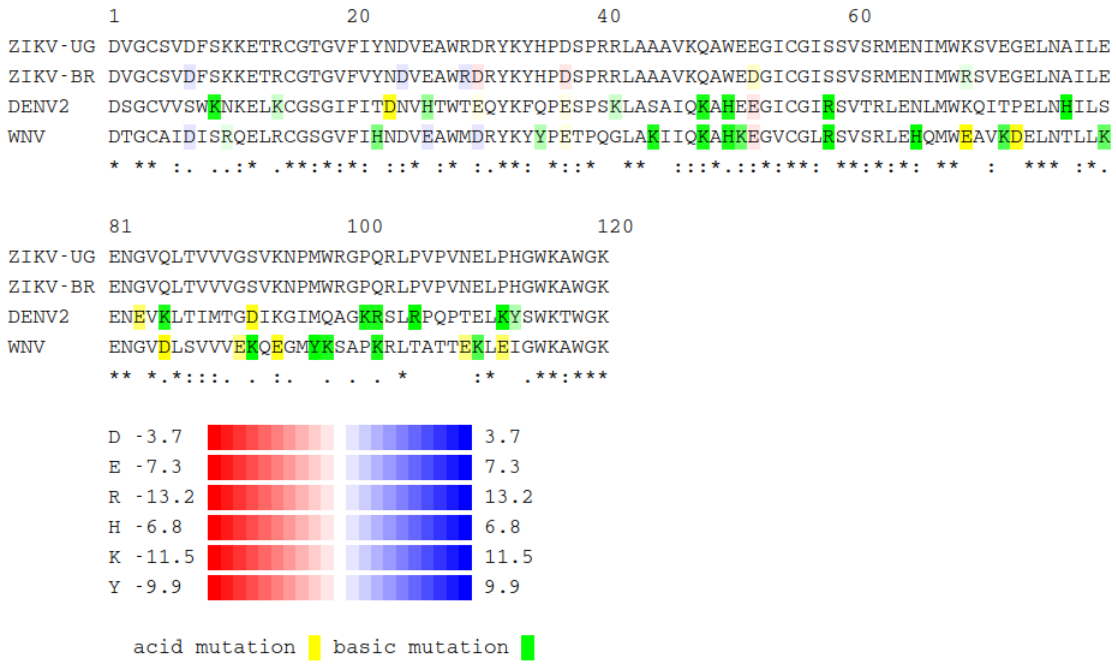


Figure 7.13.1: Alignment colored applying `nr` descriptor and the condition `Y` to separate the maximum per target-residue.

Now, the `maximum (abs)` column is displaying the maximum for each target-residue. In other words, now the values for each amino acid will be normalized according to their maximum. This list and its order will match according to the target-residues that were declared in the tuple (see Section 3.1). The alignment of the HTML file for this new condition appears in Figure 7.13.1. Now, the scale for each target-residue can be observed. A first impression may seem like there is no difference between what we got earlier (Figure 6.0.1b). However, some regions are more faint in color (see some amino

acids between positions 20 and 40 on the alignment). The intensity of the color is still not enough because the maximum values are still very high because the mutations. We can confirm this by invoking the color descriptor like this:

```
descriptor_col = { 'fac' :  [6, 20, 22, 54, 94, 46] }
```

Now we can clearly see the influence of the maximum for each target-residue (Figure 7.13.2). See how the most intense colors match with mutations.

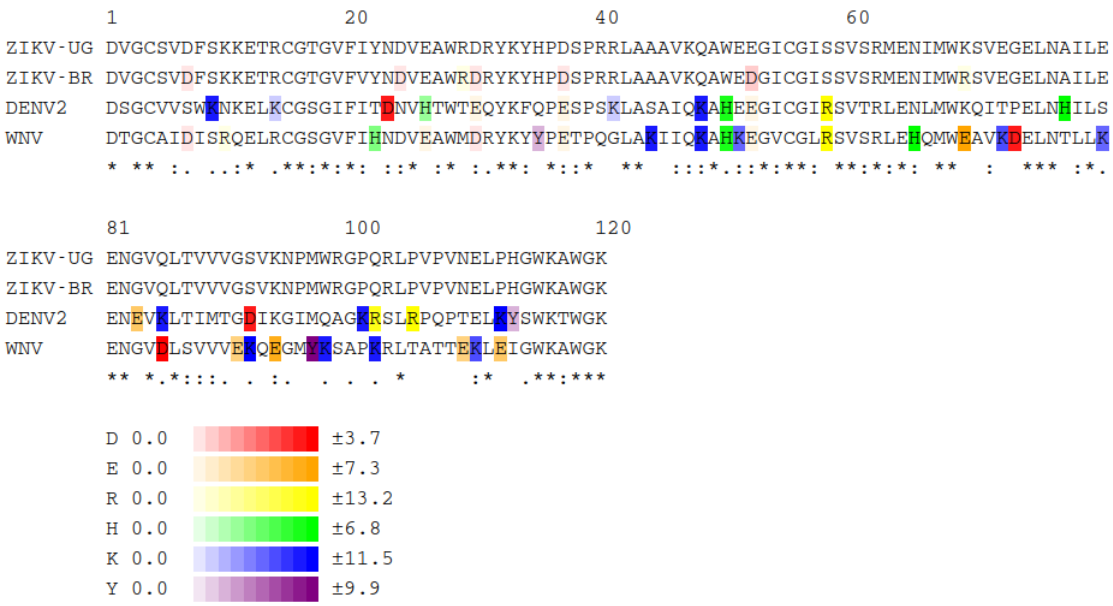


Figure 7.13.2: Alignment colored applying `nr` descriptor and the condition `Y` to separate the maximum per target-residue but using `fac` descriptor.

Finally, you can test different descriptors and observe the behavior separately based on maximums for each target-residue. The base structure for invoking these conditions is shown as follows:

```
descriptor_mxr = { 'maximum_descriptorY' }
```

```
descriptor_mxr = { 'maximum_descriptorY' : additional_parameters }
```

Note that we are considering situations where descriptors can be sets (*e.g.* `nr`, `rm`, and so on) or dictionaries (*e.g.*, `rt`, `rsa`, `rspi`, and so on), respectively.

Note: We recommend the user to test different combinations of maximum restriction and color (see Chapter 5*) descriptors, to see the impact of each case. Keep in mind the same rules for each*

maximum restriction descriptor that we saw previously

7.14 Maximum restriction descriptors dictionary (-dmx)

When it is executed the `multiple` comparison method, it is possible that you need to distinguish each reference/target(s) relationship with a specific maximum restriction descriptor. For that, we must build an array of maximum descriptors through the `dict_mxr` variable of the script "array_get.py". To build it, you should follow the same rules like for `dict_ope` (see Section 4.5). For this example, we used the same dictionary for the `multiple` method as a base (see Subsection 3.10.3). The CPRISMA maximum-restriction-array will be declared as follows[5]:

```
dict_mxr = { 'main_0' :  [ { 'rmY' } , { 'nr' } ] ,
             'main_1' :  [ { 'rspim' :  [ {'2' :  [47, 48, 50] }] } ,
                           { 'rsaY' : [ { '0' :  ['R'] } , { '3':  ['D'] } ] ,
             'main_2' :  [ { 'rt' :  [0, '>='] } ] }
```

Now we can execute the `-dmx` flag to call the array `dict_mxr`[6]:

```
cprisma -tr -ck -rf multiple -dmx
```

Your log file will show the relationships between each comparison and maximum restriction descriptors:

```
Comparing dict_ref with dict_mxr:
                                {'0':  [1, 2, 3]} ........  { 'rmY' }
                                {'0':  []} ........  { 'nr' }
                                {'1':  [0, 2]} ........  { 'rspim' :  ...  }
                                {'1':  [0, 2, 3]} ........  { 'rsaY' : ...  }
                                {'2':  [0]} ........  { 'rt' :  [0, '>='] }
```

[5] Note that we have differentiated each maximum restriction descriptor for each comparison with a different color to aid the reader.

[6] Notice that additional parameters are being implemented.

`The array of the feature 'maximum' dict_mxr is compatible with the array of comparison sequences dict_ref!`

... and a table with a summary for all maximum descriptor information per comparison:

main	comparison	descriptor	sequence	residue	position	mutation	threshold	separate
main_0	1	rmY	NaN	NaN	NaN	True	NaN	True
main_0	2	nr	NaN	NaN	NaN	NaN	NaN	NaN
main_1	3	rspim	2	NaN	[47...]	True	NaN	NaN
main_1	4	rsaY	0	[R]	NaN	NaN	NaN	True
main_1	4	rsaY	3	[D]	NaN	NaN	NaN	True
main_2	5	rt	NaN	NaN	NaN	NaN	[0, >=]	NaN

Note: If the array of `dict_mxr` *is wrongly built, all descriptors will be transformed to* `nr` *(default parameter) for each comparison and additional information about where the errors are will be displayed.*

Chapter 8

Practical Examples

The following examples bring together some of the features that we described earlier. Each is related to a previously published study [1, 2]. Furthermore, in the "examples/" directory, you will find the scripts, alignments and other files that help as the basis for achieving the desired visualizations. Many details will not be given as we wish to motivate the reader to test and explore the characteristics of each example on their own. Each case follows the next order:

1. Command line invoked.
2. Declared variables (as shown in Figure 2.2.3).
3. An image of the alignment.

Each example tries to approximate a certain image from a specific publication. In example 1 (see Section 8.1) we try to reproduce Figure 8 [1], in example 2 (see Section 8.2) Figure 5 (only chain A) [2], and in example 3 (see Section 8.3) Figure 1a [3].

Note: For more details on the methodological aspects, data collection, purpose, and others, you can find them in the respective publication.

8.1 Example 1: ΔpK_a for several $\mathbf{NS1_{ZIKV}}$

Command line:

```
cprisma -tr -ns -ck -rf multiple -va 2 -j -sco 142 -dop -dco -dvi -dmx
```

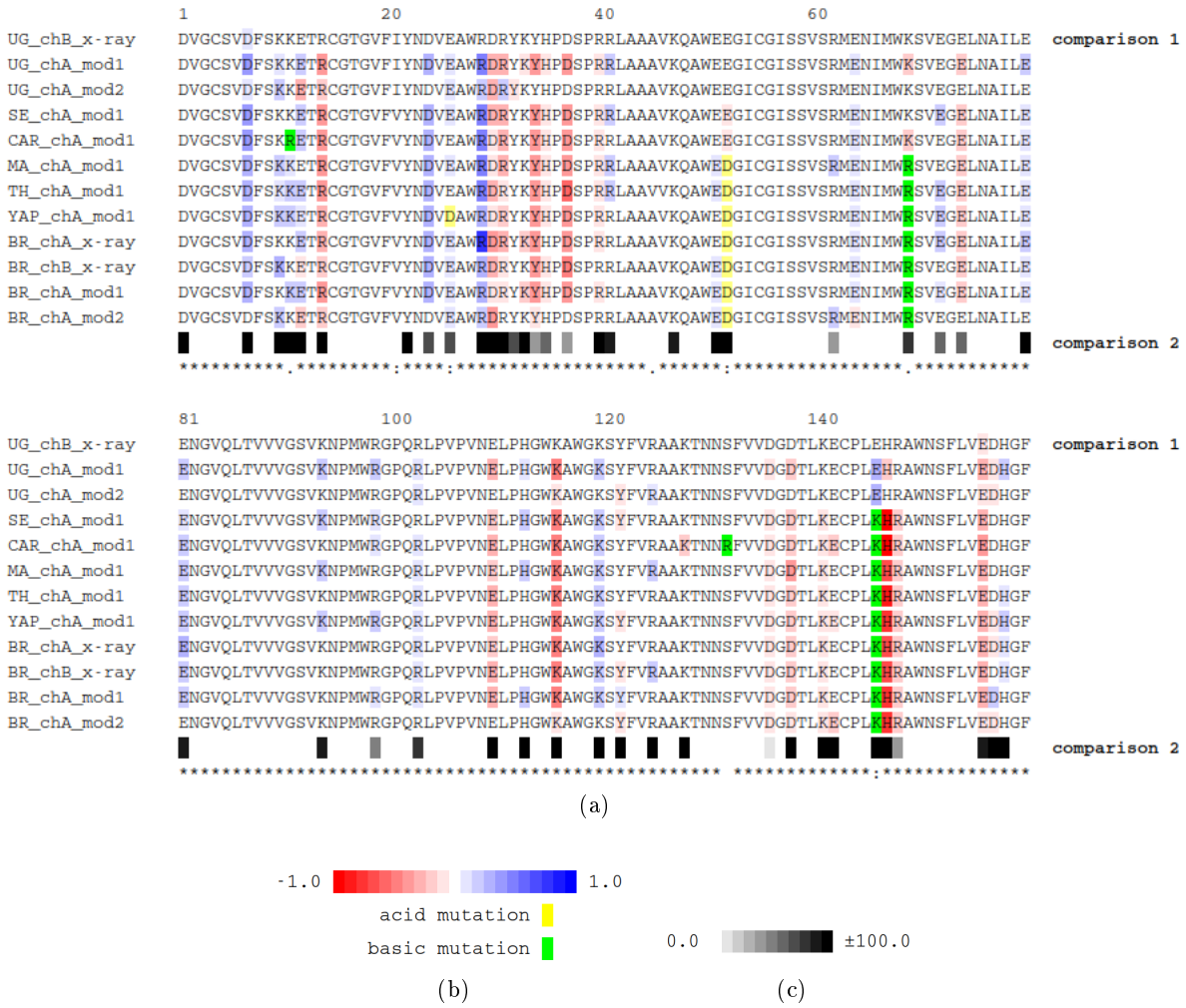


Figure 8.1.1: (a) Alignment for Example 1. Not all alignment is being shown. (b) Scale for comparison 1. (c) Scale for comparison 2.

Variables:

```
target_residues = ('D', 'E', 'R', 'H', 'K', 'Y')
```

```
name_sequence = ('UG_chA_x-ray', 'UG_chB_x-ray', 'UG_chA_mod1', 'UG_chA_mod2',
'SE_chA_mod1', 'CAR_chA_mod1', 'MA_chA_mod1', 'TH_chA_mod1', 'YAP_chA_mod1',
```

```
'BR_chA_x-ray', 'BR_chB_x-ray', 'BR_chA_mod1', 'BR_chA_mod2', '')

descriptor_ope = {'n'}

descriptor_col = { 'nc' }

descriptor_vis = { 'ReY','DeN','LeY' }

descriptor_mxr = { 'nr' }

dict_ref = { 'main_0' :  [ {'0' :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] } ] ,

             'main_13' :  [ {'13' :  [] } ] }

dict_ope = { 'main_0' :  [ { 'd' } ] ,

             'main_13' :  [ { 'n' } ] }

dict_col = { 'main_0' :  [ { 'pkac' :  True } ] ,

             'main_13' :  [ { 'ssc' } ] }

dict_vis = { 'main_0' :  [ { 'ReN','DeY','LeY' } ] ,

             'main_13' :  [ { 'ReY','DeY','LeN' } ] }

dict_mxr = { 'main_0' :  [ { 'rm' } ] ,

             'main_13' :  [ { 'nr' } ] }
```

From these input parameters, we obtain the alignment that appears in Figure 8.1.1. Note that the last element of the tuple `name_sequence` does not have a name. Due to that, the alignment will not display any name for that sequence.

8.2 Example 2: B-cell epitope predictions on $NS1_{WNV(176-352)}$

Command line:

```
cprisma -tr -ck -rf multiple -ns -va 2 -j -hc -n 176 -sco 142 -a 3 -dop -dco -dvi
```

Variables

```
target_residues = ('D', 'E', 'R', 'H', 'K', 'Y')

name_sequence = ('SASA', 'pKa_NS1 (alone)','PROCEEDpKa', 'PISA', 'DiscoTope',
'ElliPro', 'SEPPA2', 'SEPPA3', 'Consensus', '')

descriptor_ope = { 'n' }

descriptor_col = { 'nc' }

descriptor_vis = { 'ReY','DeN','LeY' }

descriptor_mxr = { 'nr' }
```

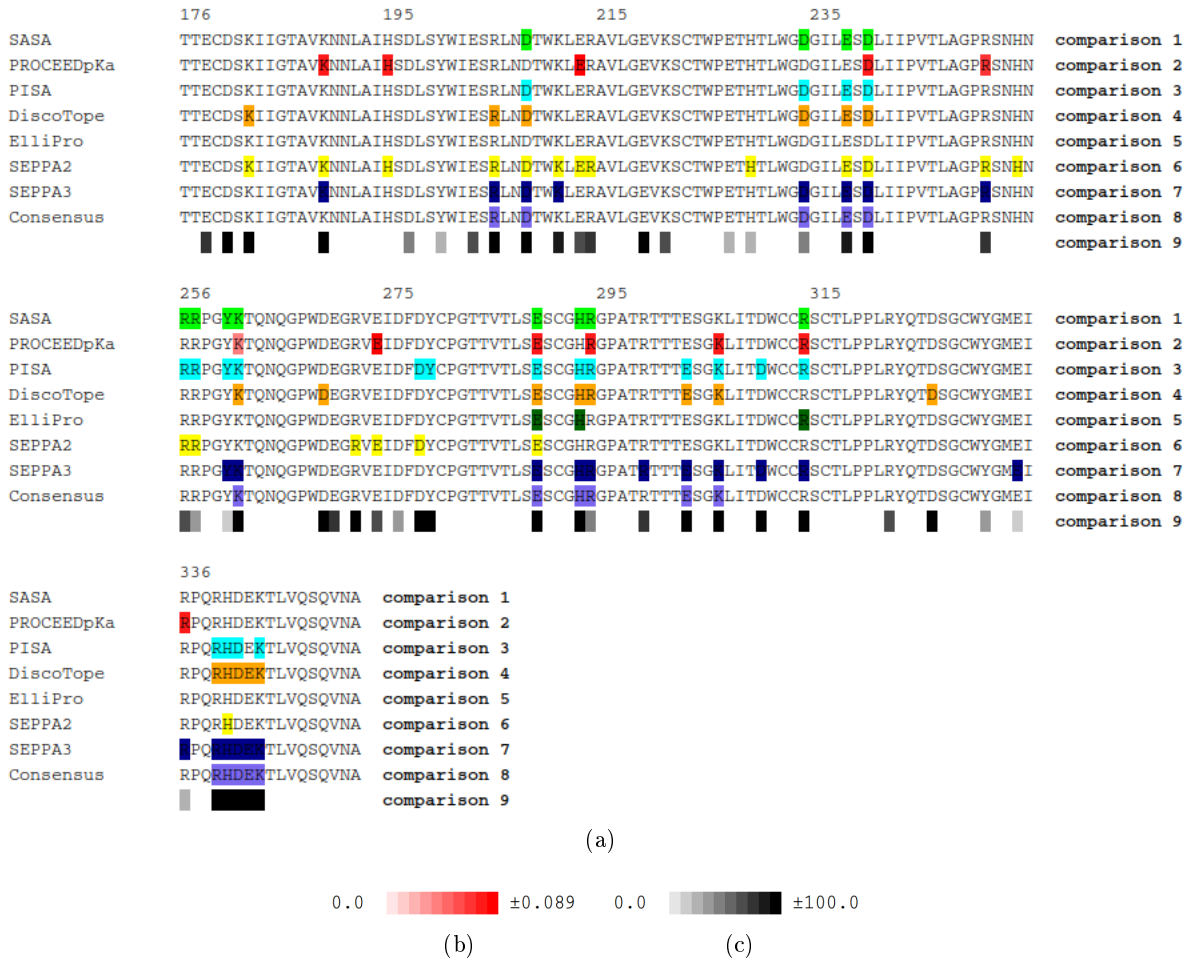


Figure 8.2.1: (a) Alignment for Example 2. (b) Scale for comparison 2. (c) Scale for comparison 9.

```
dict_ref = { 'main_0' :  [ {'0' :  [] } ] ,
             'main_1' :  [ {'1' :  [2] } ] ,
             'main_3' :  [ {'3' :  [] } ] ,
             'main_4' :  [ {'4' :  [] } ] ,
             'main_5' :  [ {'5' :  [] } ] ,
             'main_6' :  [ {'6' :  [] } ] ,
             'main_7' :  [ {'7' :  [] } ] ,
             'main_8' :  [ {'8' :  [] } ] ,
             'main_9' :  [ {'9' :  [] } ] }
dict_ope = { 'main_0' :  [ { 'n' } ] ,
             'main_1' :  [ { 'da' } ] ,
             'main_3' :  [ { 'n' } ] ,
             'main_4' :  [ { 'n' } ] ,
             'main_5' :  [ { 'n' } ] ,
             'main_6' :  [ { 'n' } ] ,
             'main_7' :  [ { 'n' } ] ,
             'main_8' :  [ { 'n' } ] ,
             'main_9' :  [ { 'n' } ] }
dict_col = { 'main_0' :  [ { 'tc' :  [54, 0, '>'] } ] ,
             'main_1' :  [ { 'tc' :  [6, 0.007, '>'] } ] ,
             'main_3' :  [ { 'tc' :  [75, 0, '>'] } ] ,
             'main_4' :  [ { 'tc' :  [20, -6.85, '>='] } ] ,
```

```
              'main_5' :  [ { 'tc' :  [64, 0, '>'] } ] ,
              'main_6' :  [ { 'tc' :  [22, 0.049, '>='] } ] ,
              'main_7' :  [ { 'tc' :  [96, 0.093, '>='] } ] ,
              'main_8' :  [ { 'tc' :  [92, 0, '>'] } ] ,
              'main_9' :  [ { 'ssc' } ] }
dict_vis = { 'main_0' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_1' :  [ { 'ReN','DeY','LeY' } ] ,
              'main_3' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_4' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_5' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_6' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_7' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_8' :  [ { 'ReY','DeN','LeY' } ] ,
              'main_9' :  [ { 'ReY','DeY','LeN' } ] }
dict_mxr={}
```

As we are implementing the `tc` color descriptor (see Section 5.8) and the numerical data is dependent on the threshold, we recommend take care with the accuracy. The output for the example 2 appear in Figure 8.2.1. To obtain this outcome, we have considered an accuracy of 3 decimal places with the `-a` flag (see Section 3.2).

8.3 Example 3: Protein structural domains for two $NS1_{ZIKV}$ and its biological interfaces

Command line:

```
cprisma -va 2 -j -ns -ck -rf multiple -mco 54 -dco -dvi
```

Variables:

```
target_residues = ()

name_sequence = (‘’, ‘ZIKV-UG’, ‘ZIKV-BR’)

descriptor_ope = { ‘n’ }

descriptor_col = { ‘nc’ }

descriptor_vis = { ‘ReY’,‘DeN’,‘LeY’ }

descriptor_mxr = { ‘nr’ }
```

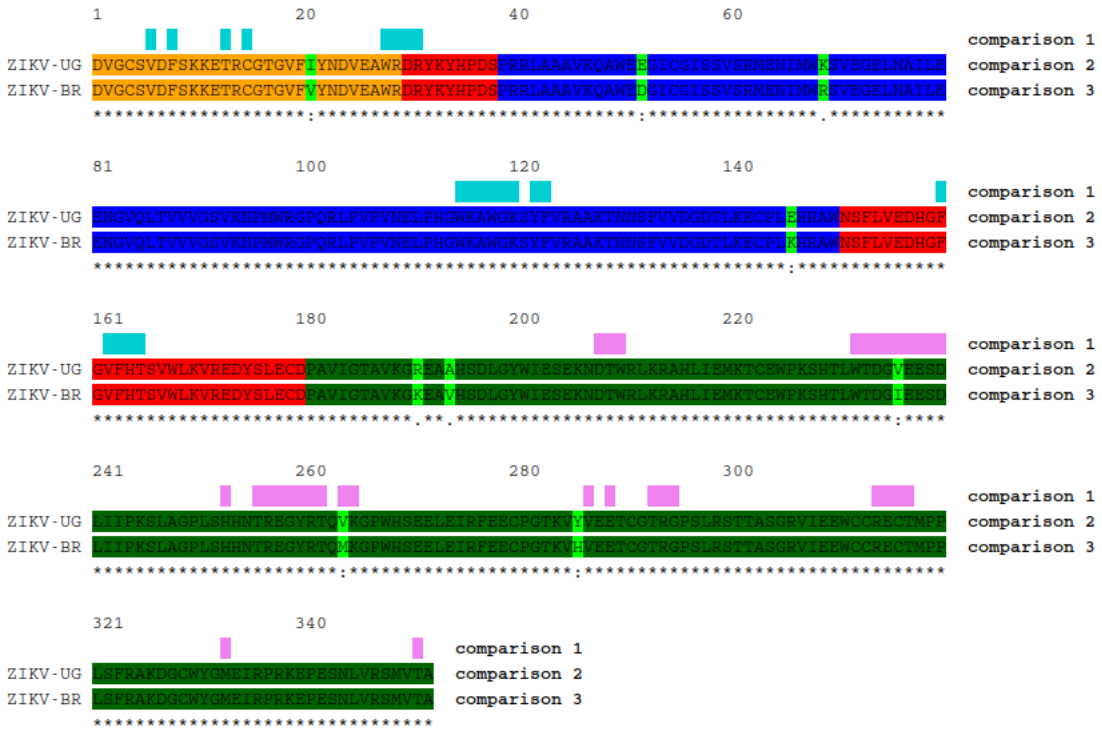


Figure 8.3.1: Alignment for Example 3.

```
dict_ref = { ‘main_0’ :  [ {‘0’ :  [] } ] ,

             ‘main_2’ :  [ {‘2’ :  [1] } ] ,
```

```
            'main_1' :  [ {'1' :  [2] } ] }

dict_ope = {}

dict_col = { 'main_0' :  [ { 'pic' :  [ { '0' :  [81, 'n', '5' ,'7', '12', '14',
                                                  '27-30', '114-119', '121-122',
                                                  '159', '161-164'] } ,
                                        { '0' :  [35, 'n', '207-209', '231-239',
                                                  '252', '255-261', '263-264',
                                                  '286', '288', '292-294',
                                                  '313-316', '332', '350'] } ] } ]
,
            'main_2' :  [ { 'pimc' :  [ { '1' :  [20, 'n', '0-28'] } ,
                                        { '1' :  [6, 'n', '29-37'] } ,
                                        { '1' :  [94, 'n', '38-149'] } ,
                                        { '1' :  [6, 'n', '150-179'] } ,
                                        { '1' :  [64, 'n', '180-351'] } ] } ] ,
            'main_1' :  [ { 'pimc' :  [ { '2' :  [20, 'n', '0-28'] } ,
                                        { '2' :  [6, 'n', '29-37'] } ,
                                        { '2' :  [94, 'n', '38-149'] } ,
                                        { '2' :  [6, 'n', '150-179'] } ,
                                        { '2' :  [64, 'n', '180-351'] } ] } ] }
dict_vis = { 'main_0' :  [ { 'ReY','DeN','LeN' } ] ,
            'main_2' :  [ { 'ReN','DeN','LeY' } ] ,
            'main_1' :  [ { 'ReN','DeN','LeY' } ] }
```

```
dict_mxr = {}
```

The output for the example 3 appears in Figure 8.3.1. See that for `pimc` descriptor to distinguish the mutations (see Section 5.7), the `main`'s `1` and `2` in `dict_ref` have the sequences exchanged and we have hidden the references with the `ReN` visualization method (see Chapter 6).

Bibliography

[1] Sergio Alejandro Poveda-Cuevas, Catherine Etchebest, and Fernando Luís Barroso da Silva. Insights into the ZIKV NS1 virology from different strains through a fine analysis of physicochemical properties. *ACS omega*, 3(11):16212–16229, 2018.

[2] Sergio Alejandro Poveda-Cuevas, Catherine Etchebest, and Fernando Luís Barroso da Silva. Identification of electrostatic epitopes in flavivirus by computer simulations: The PROCEEDpKa method. *Journal of Chemical Information and Modeling*, 60(2):944–963, 2020.

[3] Sergio Alejandro Poveda-Cuevas, Fernando Luís Barroso da Silva, and Catherine Etchebest. How the strain origin of ZIKV NS1 protein impacts its dynamics and implications to their differential virulence. *Journal of Chemical Information and Modeling*, (In press), 2021.

[4] Fernando Luís Barroso da Silva and Donal MacKernan. Benchmarking a fast proton titration scheme in implicit solvent for biomolecular simulations. *Journal of Chemical Theory and Computation*, 13:2915–2929, 2017.

[5] Fernando Luís Barroso da Silva and Luis Gustavo Dias. Development of constant-pH simulation methods in implicit solvent and applications in biomolecular systems. *Biophysical Reviews*, 9(5):699–728, 2017.

[6] Robert Edgar. MUSCLE a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5(1):113, 2004.

"One should never try to prove anything that is not almost obvious"

Alexander Grothendieck