VIPER: Virus Inhibition via Peptide Engineering and Receptor Mimicry

Anna Klingenberg & Dario Ghersi

https://github.com/A-Klingenberg/VIPER

Disclaimer and Acknowledgements

These programs are distributed in the hope that they will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for any purpose. The entire risk as to the quality and performance of the program is with the user.

VIPER was developed by Anna Klingenberg in the College of Information Science & Technology, University of Nebraska at Omaha.

Email addresses:
Anna Klingenberg: aklingenberg@unomaha.edu
Dario Ghersi: dghersi@unomaha.edu

Contents

1 — Introduction

1.1 Overview of VIPER

The VIPER suite integrates in-house as well as widely used bioinformatics tools to streamline the creation of inhibitory decoy peptides that mimic protein-protein binding sites. The VIPER program flow consists of four main steps: input, complex analysis, residue selection, and iterative improvement (see Figure 1.1). VIPER is modular and works through standardized interfaces and common file formats such as PDB files and can be extensively configured. Therefore, alternative tools can easily be integrated. Currently, VIPER takes advantage of the Rosetta suite of programs found at https://www.rosettacommons.org, as well as the PEPstrMOD web service found at https://webs.iiitd.edu.in/raghava/pepstrmod/.

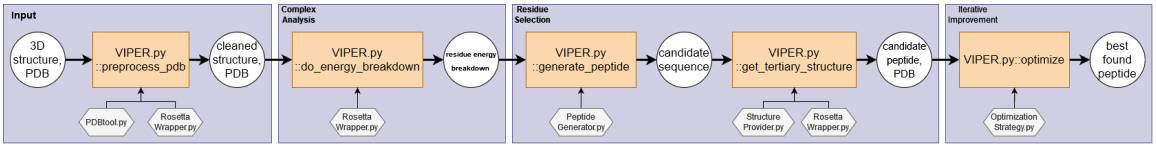


Figure 1.1: **Flowchart for VIPER, from input to iterative improvement.** VIPER takes as input a PDB of the complex of both a viral surface protein and a human receptor protein. This complex is analyzed to identify the binding site and residues involved. Using this information, a subset of residues is selected as the initial decoy peptide. The resultant peptide is then iteratively optimized.

1.2 VIPER setup & availability

The code for VIPER is available through GitHub at: https://github.com/A-Klingenberg/VIPER. VIPER depends on several Python3 packages and has other dependencies. Packages and dependencies for each step are described in detail in their respective chapters. However, utilizing the pip installation method as described in the Quick Start Guide chapter is the easiest way to get started. Although the instructions provided in each section may not account for all possible scenarios, there are lots of valuable online resources to find solutions.

1.3 Input

VIPER is capable of handling PDB files using the in-house Python component PDBtool. PDBtool provides various functions for manipulating PDB files, such as removing, renumbering, or superimposing chains.

The input PDB is reduced to the relevant chains and then checked for id conflicts. Afterwards, the residues and atoms are renumbered in an ascending manner. PDBtool provides functionality for restoring the original numbering. This preprocessed PDB file is then relaxed using the RosettaCommons software suite [Kha+11; Mag+20]. The output of this step is a PDB file that can easily be used manually, however, VIPER is able to use it directly and automatically proceed with the next step.

1.4 Complex Analysis

To ascertain which parts to build on when designing an inhibitory peptide, VIPER analyzes the per-residue interactions between the proteins in complex to identify the residues most involved in the protein-protein interactions using the Rosetta software suite [Lem+20]. VIPER can combine the results for multiple different partner chains and aggregate the results over multiple different poses. The total per-residue energies are then passed to the next step.

1.5 Residue Selection

Generating a candidate peptide from the residue energy breakdown requires a strategy for selecting specific residues based on their properties. VIPER provides two such built-in strategies, GreedyExpand and FragmentJoiner, with the latter being the default. Users may also provide their own selection strategy through the standardized interface. The FragmentJoiner, at a high level, tries to identify extended sections of connected, favorably interacting residues in the receptor protein which it then tries to combine in a manner that optimizes the total energy while taking three-dimensional constraints into consideration.

Once such a combination of original residues is identified and potentially linked using linker sequences, the tertiary structure of the resultant amino acid sequence is predicted using the PEPstrMOD web service [KGR07; Sin+15]. The predicted structure is saved as a PDB and superimposed onto the original receptor protein.

1.6 Iterative Improvement

The candidate solution from the preceding step is then subjected to an iterative improvement procedure. VIPER provides a built-in genetic algorithm optimizing for a binding energy related term and a structural stability related term, which is further explained in a following section. The user may also provide their own optimization strategy by implementing the respective interface.

When this step is completed with the default genetic algorithm, VIPER will write a list of evaluated candidates in JSON format to disk and report the best performing peptide found. PDB files for all candidates are available.

2 — Quick Start Guide

2.1 Installs

The code for VIPER is available through GitHub at: https://github.com/A-Klingenberg/VIPER. Download or clone the latest version to your location of choice.

Using git command line:

```
git clone https://github.com/A-Klingenberg/VIPER
```

If you are experiencing issues, you can try copying this command from the quick start guide on GitHub. Some people have experienced issues copying from the PDF directly, as the text might contain some signs that not every terminal can handle.
Recommended Setup:

```
# Enter VIPER directory
cd VIPER

# Create a virtual environment and activate it
python -m venv ./.venv/ # You might have to use 'python3'
source .venv/bin/activate

# Install Python library dependencies
pip install -r requirements.txt # You might have to use 'pip3'

# Modify "VIPER/config.json" to point to Rosetta install (see below) and modify
# any other settings you wish, such as how many cores to use
# Use preferred text editor (Vi, Nano, etc)
open config.json
```

In order to run VIPER, a local version of Rosetta needs to be installed. For licensing details visit

https://www.rosettacommons.org/software/license-and-download. Installation instructions can be found at https://new.rosettacommons.org/docs/latest/getting_started/Getting-Started. Rosetta should be built with MPI executables and instructions can be found at https://new.rosettacommons.org/docs/latest/build_documentation/Build-Documentation (or follow the instructions below). While there is the option to run without MPI, it is not recommend as running with MPI *significantly* speeds up computation. For more instructions on how to run `VIPER` and for a list of command line options, see chapter 4.
If MPI is not installed on your computer, you can install it using the following command (for a system with an apt installer):

```
# Linux
sudo apt install libopenmpi-dev
# Or visit: https://www.open-mpi.org/faq/?category=building

# MacOS (not officially supported by VIPER!)
xcode-select --install
brew install openmpi
# Or visit: https://www-lb.open-mpi.org/software/ompi
```

Please read below for general build options of Rosetta for Linux and MacOS systems. RosettaCommons forums are a good place to resolve compilation issues.

```
# Navigate to rosetta download
cd rosetta_src_xx/main/source
# Recommended build command
./scons.py bin mode=release extras=mpi -j<number_of_processors_to_use>

# MPI not working? Potentially need to point to mpicxx and mpicc
# Print location of mpicxx and mpicc
which mpicxx
which mpicc

# Use preferred text editor to open site.settings
open tools/build/site.settings

# Uncomment cxx and cc override xml options,
# replace with location of your devices executables
```

The configuration file for `VIPER` is at the root level of cloned repository `/VIPER/config.json`. Modify the configuration file with your preferred text editor, specifying the location of Rosetta installation folder in line 11 inside the quotation marks (`rosetta_config` → `path`). Also set the number of cores a run of a Rosetta application should use in line 14 *without* quotation marks (`rosetta_config` → `use_num_cores`).

2.2 Run VIPER

For all the steps, we assume that your working directory is "./VIPER/". The first step is to gather a PDB file of a viral surface protein in complex with a receptor protein. The example output files

assume that you are using the SARS-CoV-2 SPIKE protein in complex with the human ACE2 receptor, PDB accession code: `6m0j`.

When you run `VIPER`, there are **three** command line arguments you must pass to it. First, you need to tell `VIPER` what PDB file to use. Second, you need to tell it what the chain identifier for the viral surface protein is using the `--vsp_chain` argument. Third, you need tell it what the chain identifier for the receptor protein is using the `--partner_chain` argument. You can gather this information from the PDB file itself by examining it in your text editor of choice. You may provide the chain identifiers as part of the `config.json` file in lines 7 and 8 instead of as command line arguments, and you may specify multiple chains by simply appending the chain identifiers, such as 'AB'. Keeping with the `6m0j` example, the command looks like the following, assuming that the PDB file of the complex is saved in the root '`VIPER/`' directory.

```
1 python3 -u main.py 6m0j.pdb --vsp_chain E --partner_chain A &> OUTFILE &
```

It is recommended to launch the application in a background process using the trailing ampersand. Additionally, while `VIPER` logs all its messages to a dedicated file, some tools and libraries may emit their own messages. To capture these, you may append the '`&> OUTFILE`' part as shown above, which will capture these messages and write them to a file called '`OUTFILE`' in the './VIPER/' directory.

Running `VIPER` in full will usually take a few hours, depending on the compute resources available. During this time, you can periodically check the progress by checking the most current few lines of the log file using the following command on Linux.

```
1 # This assumes the default logging folder.
2 # If you have changed this in the config.json, you have to change the path here
3 cat Logs/log.txt | tail -n 30
```

The command above will print the 30 most recent (last) lines of the log file. If you want to change the number of lines shown, adjust the number at the end of the command.

Alternatively, you can check the output folder for PDB files created by `VIPER` and other software tools. The standard output path, if you have not changed it, is 'output'. To recursively list the content of this folder and all subfolders, you may use the following command on Linux.

```
1 # This assumes the default output folder.
2 # If you have changed this in the config.json, you have change the path here
3 ls output/ -R
```

The output folder structure generally looks like the following, assuming you haven't changed the default config values and `VIPER` has been run to completion.

- output/
  - candidates/
    - 0/
      - candidate_0_relax_ensemble
      - candidate.pdb — This is the predicted structure of the initially generated peptide.
      - GA/ — All GA output is saved here.
        - gen0_*SEQUENCE*/ — *SEQUENCE* is the peptide sequence in single letter format.

- base.pdb The predicted structure of this peptide.
- base_upd_chains.pdb
- best_complex_orinum.pdb The lowest energy pose of the peptide+VSP complex, with original numbering and chain IDs.
- best_complex.pdb
- *SEQUENCE*_aligned.pdb
- *SEQUENCE*_aligned_renumbered.pdb
- *SEQUENCE*_aligned_vsp_concat.pdb
- interface_score.sc Contains Rosetta metrics in regard to the binding interface.
- reb_score.sc Residue energy breakdown for the peptide+VSP complex.
- relax/
  - complex/ Holds all relaxed poses of the complex generated by Rosetta.
    - *SEQUENCE*_aligned_vsp_concat_relax_0001.pdb
    - ...
    - score_relax.sc List of scores for all poses.
  - SCORE Only contains the total calculated score for this peptide.
  - score_log.txt A log file for the scoring process for this peptide.
  - warnings.json If it exists, holds all contact checking warnings.
- gen0_*SEQUENCE*/
- gen1_*SEQUENCE*/
- ...
- scores.json This is a collection of all evaluated peptides and their scores.
- vsp.pdb This is a PDB of only the viral surface protein.

- reference/
  - *INPUT*_renum.pdb
  - *INPUT*_renum_relaxed.pdb This is the lowest energy pose found and will be used as the actual reference PDB.
  - intermediary/ All relaxed poses of the input PDB are saved here.
    - *INPUT*_renum_relax_pinned_0001.pdb
    - ...
    - score_relax.sc The scores for all poses are listed in this file.
  - reference_ensemble
- rosetta_output/
  - docking/
  - prepack/
  - refine/
  - relax/
  - residue_energy_breakdown/
    - reference_pdb/
      - energy_breakdown_*INPUT*_renum_relaxed.out This is the per residue energy breakdown of the input complex.
  - run_configs/
    - flag_*APPLICATION*_*SEQUENCE-or-ID* These are config flags for running the specific Rosetta application.
    - ...

3 — Under the Hood

The following chapter will give a more in-depth overview of the internal workings of VIPER.

3.1 The Input Step

During the input step, the PDB file of the protein-protein complex is prepared for usage.

This includes two major steps: PDB cleaning and relaxation. First, VIPER utilizes the built-in PDBtool to perform several preprocessing steps. PDBtool starts with removing all chains that are not specified as being either a partner chain or a viral surface protein chain. This behaviour may be turned off via a config option ("remove_other_chains"). Following this, the chains get reordered, so that the partner chain(s) precede the viral surface protein chain(s), after which all atom and residue ids within the PDB get renumbered in an ascending manner, starting from 1. This then concludes the PDB preprocessing, after which the resultant structure is relaxed using the Rosetta software suite. This relaxation is a specifically configured version of the FastRelax protocol [Kha+11; Mag+20], where most atom positions are pinned in place through constraints that penalize large translations and rotations, in order to conserve more of the original structural binding information. The specific configuration can be seen in `RosettaWrapper::Flags::relax_pinned_positions`. With default settings a total of *100* relaxed structures will be generated, although this can be configured using the "rosetta_config.prerelax_complex_runs" setting. All generated structures a scored using the Rosetta scoring function, and the structure with the lowest energy is then used as a reference structure for the rest of the program.

Notably, VIPER uses a simple heuristic to identify and reuse relaxed reference structures. As intermediate and processed PDB files are saved to the disk with specific suffixes appended to the input PDB name, if a PDB file exists in the reference output folder that has the suffixes of the final relaxed structure appended to the name of the current input PDB file, VIPER will reuse this file if not otherwise configured. The corresponding setting in the `config.json` file is called `reuse_preprocessed`, with the default allowing the reuse of preprocessed PDB files.

3.2 The Complex Analysis Step

Once the complex structure is prepared, VIPER proceeds with analyzing the binding interaction using the `residue_energy_breakdown` Rosetta application [Lem+20]. This application enumerates the per-residue interactions between all residue-residue pairs in the complex. The application is executed via the `RosettaWrapper`, which can execute arbitrary Rosetta applications with arbitrary options. In this case, the specific configuration can be seen in `RosettaWrapper::Flags::residue_energy_breakdown`.

What this application outputs is a structured text file, with one interaction per line. Each line contains the residue in question and its partner, if applicable (so not a onebody interaction), the name of the input structure, and the Rosetta score terms. An example of a line in this output file is given below.

```
SCORE: pose_id  resi1  pdbid1  restype1  resi2  pdbid2  restype2  fa_atr ... total
    ↪ description

    ...

SCORE: test.pdb  5  23A  GLU  740  475E  ALA  -0.172 ... -0.204   test.pdb_5_740
```

In the example above, the individual score terms have been abbreviated with the '...' placeholder. Additionally, only an example of a twobody interaction is shown. Here, the 23rd residue which is situated on chain A, a glutamic acid, interacts with the 475th residue which is situated on chain E for a total energy of -0.204. This information is written to disk in the `output/rosetta_output/residue_energy_energy_breakdown/reference_pdb/` folder.

Using this information, VIPER constructs `RosettaWrapper::REBprocessor::Node` objects for every residue in the structure, which save the residue identity, other residues they interact with and the strength of the interactions, as well as a reference to the neighboring residues on the same chain. Notably, VIPER allows for processing more than structure, or pose, at a time. If the `rosetta_config.reb_only_use_best` setting is not set to `True` (not the default), VIPER will average the interaction energies over all relaxed structures. Onebody interactions will be ignored, as only residue-residue interactions are relevant for VIPER.

3.3 The Residue Selection Step

With the information on how strongly each residue in the receptor interacts with the residues in the viral surface protein, VIPER can then derive a candidate inhibitory peptide by selecting a subset of receptor residues. How to select residues isn't trivial though. VIPER provides two built-in strategies, `modules:stages::PeptideGenerator::GreedyExpand` and `modules:stages::PeptideGenerator::FragmentJoiner`, with the latter being the default, as well as a supported mechanism of writing your own residue selection strategy. All residue selection strategy must subclass `modules::interfaces::ResSelectionStrategy::ResSelectionStrategy`. If you wish to write your own strategy, implement `custom_funcs::CustomSelectionStrategy` and set `peptide_generator.custom_strategy` to `true`.

Both strategies have the same idea at its core, which is growing larger contiguous subsets from individual, strongly favorable interacting residues. How they implement this procedure is what distinguishes them. You can switch between them by either entering "fragment_joiner" or "greedy_expand" for the `peptide_generator.use_strategy` configuration option. To configure

the maximum length of the peptide you want to generate, set `peptide_generator.max_length` to the maximum number of residues. Note that this is just an upper bound and doesn't guarantee that the recommended peptide will be exactly this many residues long.

3.3.1 GreedyExpand

`GreedyExpand` first sorts all `Node` objects by the sum of interaction strengths with the viral surface protein. Then, working from the strongest interacting `Node` to the weakest interacting one, the algorithm tries to extend the current selection of nodes by extending it on either side if the neighboring `Node` meets certain criteria. Simplified pseudocode for the function that grows the current selection is shown in algorithm 1. Be aware that this version omits several checks and steps for the sake of explaining the underlying idea more concisely. The actual code can be found under `modules::stages::PeptideGenerator::GreedyExpand::reduce`.

Algorithm 1 `GreedyExpand` grow selection function (simplified)

Constants:
max_length: Maximum allowed peptide length.
max_side_extension: Maximum allowed length to extend selection in either direction.
vsp_chain: Viral surface protein chain.
energy_thresh: Energy threshold below which a residue should be included.
Arguments:
curr_sel: The current selection. This is a list of *Node* objects.
node: The *Node* object to be considered for inclusion in the selection.
curr_depth: The distance to the start residue in number of residues.

function _INCLUDE(curr_sel: list, node: Node, curr_depth: int)
    if $LEN(curr\_sel) \geq max\_length \ \vee \ node.STRENGTH\_TO(vsp\_chain) \geq 0$ **then**
        return ▷ *Energies below 0 attract, energies above repulse.*
    else if $node.STRENGTH\_TO(vsp\_chain) < energy\_thresh \ \wedge \ curr\_depth \leq max\_side\_extension$ **then**
        if $node \notin curr\_sel$ **then**
            $curr\_sel \leftarrow curr\_sel + \{node\}$
        $prev \leftarrow node.neighbor\_prev$
        $next \leftarrow node.neighbor\_next$
        if $prev.STRENGTH\_TO(vsp\_chain) > next.STRENGTH\_TO(vsp\_chain)$ **then**
            _INCLUDE($curr\_sel$, $next$, $curr\_depth + 1$)
            _INCLUDE($curr\_sel$, $prev$, $curr\_depth + 1$))
        else
            _INCLUDE($curr\_sel$, $prev$, $curr\_depth + 1$))
            _INCLUDE($curr\_sel$, $next$, $curr\_depth + 1$))
    else
        return

The function outlined above is called for the most favorably interacting residue, i. e. the one with the lowest total interaction energy for the viral surface protein. This is then repeated with the next most favorably interacting residue that hasn't been included yet until the number of selected

residues reaches the maximum allowed length of the peptide. Once this point is reached, the selected residues are sorted in the order of their id, so that they are in the same order they are in the receptor protein, and any necessary linkers will be added, if configured.

You can replace this inclusion function with your own by implementing `custom_funcs::greedy_expand_node_inclusion` and setting the option `peptide_generator.greedy_expand.custom_func` to `True`. `VIPER` also exposes many configuration settings to customize the behaviour of `GreedyExpand`, which are outlined below.

Name	Type	Description	Default
`peptide_generator.greedy_expand.`			
`energy_thresh`	float	The total interaction energy a residue has to meet or be lower than to be accepted as an extension of the current selection.	-0.1
`max_side_extension`	integer	How many residues to include at most in each direction starting from the initial residue to grow the selection from.	2
`ignore_neighbors`	boolean	Whether to never include any neighboring residues.	False
`always_include_direct_neighbors`	boolean	Whether to always include the direct neighbors of a starting residue. This supersedes `ignore_neighbors`	False
`custom_func`	boolean	Whether to use the custom inclusion function from `custom_funcs.py`	False

3.3.2 FragmentJoiner

While `GreedyExpand` is offered in `VIPER`, the default residue selection strategy is `FragmentJoiner`, a more complex strategy that respects three-dimensional constraints.

As a high level synopsis, `FragmentJoiner` performs a forward scan through the receptor to identify connected sections of favorably interacting residues. It is tolerant to small subsections with unfavorably interacting residues and can bridge them, if configured. Thus, it generates a list of so called 'fragments', which are contiguous subsequences of residues that interact favorably with the viral surface protein. It then determines the combination of fragments that minimizes the total interaction energy (lower is better, more attractive interactions). Notably, it records the 3D coordinates of both the N-terminus and C-terminus end of each fragment and only considers fragment combinations where the distance between different types of termini is smaller than or equal to a configurable length. By default, `VIPER` set this length to 4.0 ångström times the number of residues in the linker to approximate the length of the backbone. By using this method, a conformation where all fragments are at their original positions from the receptor protein is theoretically possible, while combinations where fragments can't all reach their original positions at the same time are avoided. `FragmentJoiner` can of course be extensively configured.

To provide an intuition of how the `FragmentJoiner` procedure works, the following algorithms show a simplified pseudocode version of the `FragmentJoiner` procedure.

Algorithm 2 shows the logic behind deciding whether a residue should be included in the current fragment or not. Note that several parts are omitted in favor of clarity. The actual code can be found under `modules::stages::PeptideGenerator::FragmentJoiner::reduce`. In short, a residue

Algorithm 2 `Fragmentjoiner` residue inclusion logic (simplified)

Constants:
vsp_chain: Viral surface protein chain.
abs_criterion: Whether the absolute energy increase criterion should be used.
min_abs_decrease: By how much the the total fragment energy has to decrease in absolute terms to include the residue.
rel_criterion: Whether the relative energy increase criterion should be used.
min_rel_increase: By how much percent the absolute total fragment strength has to increase to include the residue.
strict: Whether both criteria need to be True to include the residue in question.
Arguments:
node: The *Node* object to be considered for inclusion in the selection.
curr_strength: The strength of the current fragment being considered.
add_to_strength: A modifier to add to the score of the current `Node`.

function _INCLUDE(node: Node, curr_strength: float, add_to_strength: float)
    $n\_strength \leftarrow node.STRENGTH\_TO(vsp\_chain)$
    if $n\_strength = 0$ **then**
        return *False* ▷ *The residue in question does not interact with vsp_chain.*
    $abs\_fulfilled \leftarrow False$
    if $abs\_criterion \land add\_to\_strength + n\_strength < min\_abs\_decrease$ **then**
        $abs\_fulfilled \leftarrow True$
    $rel\_fulfilled \leftarrow False$
    if *rel_criterion* **then**
        $new\_strength \leftarrow curr\_strength + add\_to\_strength + n\_strength$
        $rel\_change \leftarrow \frac{new\_strength - curr\_strength}{curr\_strength}$
        if $curr\_strength > 0$ **then**
            ▷ *If current total fragment energy is positive, decreasing the total energy is desirable. Therefore, invert percent change.* ◁
            $rel\_change \leftarrow -rel\_change$
        if $rel\_change \geq min\_rel\_increase$ **then**
            $rel\_fulfilled \leftarrow True$
    return $((strict \land abs\_fulfilled \land rel\_fulfilled) \lor (\neg strict \land (abs\_fulfilled \lor rel\_fulfilled)))$

is included if the inclusion would lower the total energy of the current fragment. `FragmentJoiner` can be configured to either have the residue lower the total energy by at least a specific absolute amount, increase the absolute total energy by a set percentage (i. e. $10\% : -10.0 \rightarrow -11.0$), or both, in order to be added to the current fragment. To determine the actual fragments, `FragmentJoiner` performs a forward scan through the receptor protein and starts a fragment whenever it first encounters a residue that interacts favorably with the viral surface protein. From there on, all following residues are tested until algorithm 2 returns that a residue should not be added to the current fragment. `FragmentJoiner` possesses a configurable lookahead window to bridge small segments that don't interact favorably with the viral surface protein. If, however, no favorably interacting residue can be

found even in the lookahead window, the fragment is terminated and saved and a new fragment is started with the next residue down the chain that interacts favorably with the viral surface protein.

Algorithm 3 `Fragmentjoiner` fragment construction logic (simplified)

Constants:
vsp_chain: Viral surface protein chain.
nodes: A list of all nodes in the receptor protein.
lookahead_range: How many further residues should be checked when an unfavorably interacting residue is encountered.

$fragments \leftarrow \{\}$
$curr\_fragment \leftarrow \{\}$
$lookahead\_buffer \leftarrow \{\}$
$fragment\_strength \leftarrow -0.00000001$ ▷ *Use an ε to prevent DIV/0*
for $index = 1, \ldots, LEN(nodes)$ **do**
    if $nodes[index] \in curr\_fragment$ **then** ▷ *Residue is already added, skip to next residue.*
        continue
    $added\_residue \leftarrow False$
    $lookahead\_strength\_buf \leftarrow 0.0$
    for $offset = 1, \ldots, lookahead\_range$ **do**
        if $LEN(curr\_fragment) = 0 \wedge offset = 0 \wedge nodes[index + offset].STRENGTH\_TO(vsp\_chain) \geq 0$ **then**
            ▷ *Don't do lookahead if we'd be starting the fragment from a residue that wouldn't be included anyway.* ◁
            break
        if _INCLUDE($nodes[index + offset]$, $fragment\_strength$, $lookahead\_strength\_buf$) **then**
            $added\_residue \leftarrow True$
            for $r \in lookahead\_buffer$ **do**
                $curr\_fragment \leftarrow curr\_fragment + \{r\}$
                $fragment\_strength \leftarrow fragment\_strength + nodes[index + offset].STRENGTH\_TO(vsp\_chain)$
                break
        else ▷ *Temporarily store current residue and look ahead.*
            $lookahead\_strength\_buf \leftarrow lookahead\_strength\_buf + nodes[index + offset].STRENGTH\_TO(vsp\_chain)$
            $lookahead\_buffer \leftarrow lookahead\_buffer + \{nodes[index + offset]\}$
    $lookahead\_buffer \leftarrow \{\}$
    if $added\_residue$ **then** ▷ *We extended the fragment during this steop, so we may continue.*
        continue
    else if $LEN(curr\_fragment) > 0$ **then** ▷ *We have a non-empty fragment, but were not able to extend it in this step.*
        $fragments \leftarrow fragments + \{curr\_fragment, fragment\_strength\}$
        $curr\_fragment \leftarrow \{\}$
        $fragment\_strength \leftarrow -0.00000001$

Finally, `FragmentJoiner` exhaustively enumerates every fragment combination that has the total number of residues not exceed the maximum and selects the combination that minimizes the sum of all fragment energies. Another constraint that is placed on fragment combinations is that for each combination the α-carbons from the N-terminus in one fragment and the α-carbon from the C-terminus in the other fragment must be no more than a set distance apart. This distance is a configurable ångström distance times the number of residues in the inter-fragment linker. Notably, if `FragmentJoiner` identifies the fragment whose inclusion would improve the total strength the most but that fragment is too large to be fully joined, `FragmentJoiner` joins the biggest subset of residues it can without exceeding the peptide length limit starting from the connecting terminus.

As with `GreedyExpand`, `FragmentJoiner` offers many configuration options outlined below.

Name	Type	Description	Default
`peptide_generator.fragment_joiner.`			
`use_abs_increase`	boolean	Whether the absolute energy change criterion should be used for the residue inclusion logic.	True
`use_rel_increase`	boolean	Whether the relative energy change criterion should be used for the residue inclusion logic.	False
`min_abs_increase`	float	The amount the total fragment strength has to change by if the residue is to be included.	-0.2
`min_rel_increase`	float	The relative amount the total fragment strength has to change by if the residue is to be included. A value of 0.1 means that the absolute value of the strength must increase by at least 10% if the current strength is negative, or it must decrease by at least 10% if the current strength if positive.	0.1
`lookahead`	integer	How many residues to look ahead for if an unfavorably interacting residue is encountered.	2
`pad_lone_residues`	boolean	Whether to add neighboring residues if a fragment consists of only a single residue. Their interaction energies with the viral surface protein *will* be added to the total fragment strength!	True
`lone_residue_pad_range`	integer	How many neighbors on each side to add to the single residue fragment.	1
`penalize_lone_residues`	boolean	Whether to apply a score penalty to fragments consisting of a single residue. This penalty will be applied *after* any padding residues and their energies have been added to the total.	True
`lone_residue_penalty`	float	The percentage-based penalty to apply to the total strength single residue fragments.	0.5

`linker_stretch_factor`	float	The length in ångström of each residue in the linker when checking the inter-fragment distances. If you don't have a linker configured, this value will be used exactly. If you don't want to use a linker and/or want to penalize fragments on a sliding scale instead of flatly not using fragments beyond a certain distance, you may use the old combination method below.	4.0
`old_frag_combiner`	boolean	Enables the old fragment combiner. This method applies a configurable percentage-based penalty based on each fragment's distance to the fragment with the best (lowest) interaction strength.	False
`join_distance_penalty`	float	After how many ångström distance to the fragment with the lowest energy to start applying a score penalty.	4.0
`join_penalty_factor`	float	The percentage modifier to apply to the fragment strength per ångström distance to the best scoring fragment.	0.05
`length_flexibility`	integer	A flexible budget to exceed the peptide length limit, if necessary.	0
`custom_func`	boolean	Whether to use the custom inclusion function from `custom_funcs.py`	False

3.3.3 Length Damping

One part that has been omitted from the algorithms above is the fact that you can penalize or encourage the formation of long fragments through a mechanism called 'length damping'. As both `GreedyExpand` and `FragmentJoiner` have a way of tracking the size of the subsections they are currently expanding, it is possible to apply a progressive score modification based on the length of the current subsection/fragment. You can select either a linear or quadratic modifier and configure it to your liking. Both work by interpolating between a start modifier at a set length and an end modifier at a set length.

The equation below shows the calculation of the percentage-based modifier for the linear version.

$$f^{-}_{\text{lin}}(l_{\text{curr}}) = \max(\max(0, b_{\text{end}}), (b_{\text{init}} - \min(\max(0, l_{\text{curr}} - l_{\text{min}}), l_{\text{max}}) \cdot s) \tag{3.1}$$

$$f^{+}_{\text{lin}}(l_{\text{curr}}) = \min(\max(0, b_{\text{end}}), (b_{\text{init}} - \min(\max(0, l_{\text{curr}} - l_{\text{min}}), l_{\text{max}}) \cdot s) \tag{3.2}$$

with

$$l_{\text{curr}} = \text{Current length of sequence} \tag{3.3}$$

$$l_{\text{min}} = \text{Length after which to start applying damping} \tag{3.4}$$

$$l_{\text{max}} = \text{Length after which to stop applying damping} \tag{3.5}$$

$$b_{\text{init}} = \text{The damping factor with which to start} \tag{3.6}$$
$$b_{\text{end}} = \text{The damping factor on which to end} \tag{3.7}$$
$$s = \begin{cases} \text{config value}, & \text{if set by user} \\ \frac{b_{\text{init}} - b_{\text{end}}}{l_{\max} - l_{\min}}, & \text{otherwise} \end{cases} \tag{3.8}$$

To provide a visual intuition, figure 3.1 provides a plot of the linear length damping function for an example configuration.

Figure 3.1: Linear damping example with $b_{\text{init}} = 1, b_{\text{end}} = 0.2, l_{\min} = 1, l_{\max} = 3$.

In this case, a progressive penalty is applied, so f^{-}_{lin} is used. The first residue in a fragment would have its full interaction strength, as here $f^{+}_{lin}(1) = 1.0$ or 100%. The second residue's interaction energy would then be only 60% of the actual value and the third residue's interaction energy would then be reduced to 20% of the original value. Alternatively, you can use a quadratic penalty, which fits a parabola between the start and end points. The mathematical definition is given below.

$$f^{-}_{\text{quad}}(l_{\text{curr}}) = \max(b_{\text{end}}, (-1 \cdot (\max(l_{\text{curr}}, l_{\min}) - l_{\min})^2) \cdot \max\left(0, \frac{b_{\text{init}} - b_{\text{end}}}{(\max(l_{\min} + 0.1, l_{\max}) - l_{\min})^2}\right) + b_{\text{init}}) \tag{3.9}$$
$$f^{+}_{\text{quad}}(l_{\text{curr}}) = \max(b_{\text{init}}, \min(b_{\text{end}}, (\max(l_{\text{curr}}, l_{\min}) - l_{\min})^2) \cdot \max\left(0, \frac{b_{\text{end}} - b_{\text{init}}}{(\max(l_{\min} + 0.1, l_{\max}) - l_{\min})^2}\right) + b_{\text{init}}) \tag{3.10}$$

with

$$l_{\text{curr}} = \text{Current length of sequence} \tag{3.11}$$
$$l_{\min} = \text{Length after which to start applying damping} \tag{3.12}$$
$$l_{\max} = \text{Length after which to stop applying damping} \tag{3.13}$$
$$b_{\text{init}} = \text{The damping factor with which to start} \tag{3.14}$$
$$b_{\text{end}} = \text{The damping factor on which to end} \tag{3.15}$$

A visual example is given in figure 3.2, this time with a progressive bonus (f^{+}_{quad})being applied.

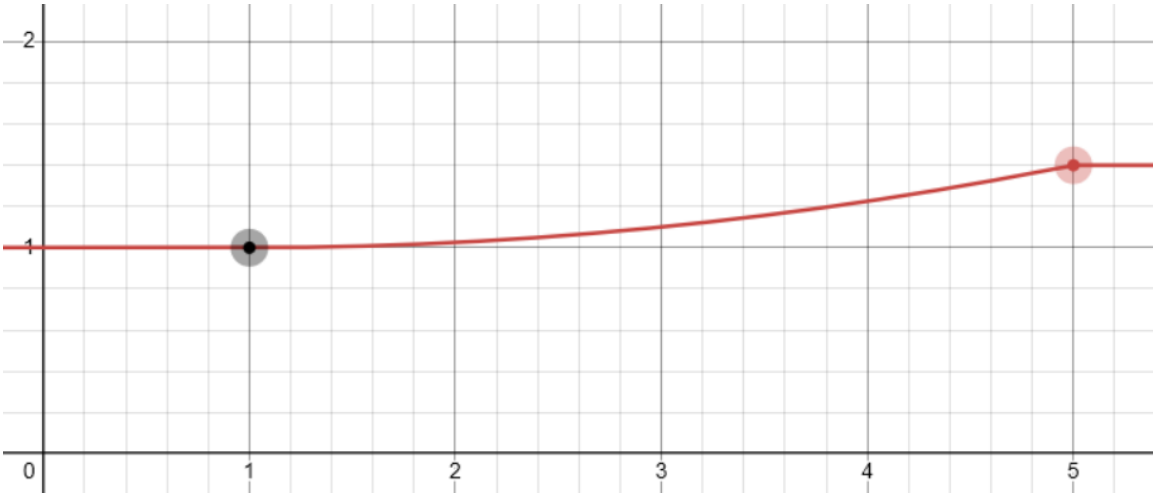


Figure 3.2: Quadratic damping with $b_{\text{init}} = 1, b_{\text{end}} = 1.4, l_{\text{min}} = 1, l_{\text{max}} = 5$.

You can extensively configure length damping via the configuration option outlined below.

Name	Type	Description	Default
`peptide_generator.`			
`length_damping`	boolean	Whether to apply length damping.	False
`length_damping_mode`	{linear ∨ quadratic}	What type of line / curve to fit between the start and end points of the damping modifier calculation.	quadratic
`length_damping_min_length`	integer	The number of residues from the start of the fragment where no energy modification should be done.	2
`length_damping_max_length`	integer	The number of residues from the start at which to reach the final energy modifier.	7
`length_damping_initial_mult`	float	The percentage-based modifier that should be applied to all residues up to `length_damping_min_length`.	1.0
`length_damping_final_mult`	float	The percentage-based modifier that should be applied to all residues including and following `length_damping_max_length`.	0.2
`length_damping_linear_stepping`	{float ∨ String}	By how much to modify the output of the function per additional residue (whole number). By improperly setting this, you can create a function that is not continuous! You can use an empty string if you want to let `VIPER` automatically calculate the correct stepping.	""

3.3.4 Adding Linkers

Once the residues to be included are identified, it may be the case that they do not form an uninterrupted chain of residues. In this case, it is possible to insert a customizable linker in each gap between connected fragments of residues.

You can specify the linker you want to use by modifying `peptide_generator.linker`, the default is two glycines ("GG"). It is important that you use the **single-letter notation** of amino acids here. There are two more relevant settings here, `peptide_generator.linker_oversize_policy` and `peptide_generator.linking_force_length_limit`. The former determines what happens when the linker is longer than the gap between fragments. For example, if there exist two fragments with a one residue gap in between them but the linker consists of two glycine residues, `VIPER` can either truncate the linker to the correct length (in this case use only the first glycine), which corresponds to the setting value `truncate`, insert the full-length linker anyway (in this case use both glycines), which corresponds to the setting value `ignore`, or skip inserting the linker (in this case don't use any glycines), which corresponds to the setting value `skip`. The latter determines what to do if inserting the linker would cause the peptide length to exceed the limit. If set to `True`, `VIPER` will include all residues up to the length limit, emit a warning, and return prematurely. Vice versa, if set to `False`, `VIPER` will ignore the length limit when adding linkers.

3.4 The Iterative Improvement Step

`VIPER` includes an optional stage where it tries to optimize the candidate peptide proposed by the residue selection strep. It does so via a bespoke genetic algorithm implementation.

3.4.1 Genetic Algorithm Architecture

Figure 3.3 provides a schematic overview of the genetic algorithm (GA) architecture.

The GA is capable of evolving multiple separate populations and optionally merging them after a set amount of generations. The overall procedure consists of the following steps:

1. Score all individuals.
2. Select parents.
3. Generate offspring from parents via the crossover operator.
4. Add offspring and parents to the next population.
5. Pad the next population to the size of the current population using random mutants of the current population.

The GA is highly configurable and you may replace it with another optimization strategy, so long as it implements `modules::interfaces::OptimizationStrategy::OptimizationStrategy`. All code for the GA is located in `modules::stages::optimize::GAStrategy.py`.

3.4.2 Scoring

As scoring is an integral part of a GA, a quick overview of the score function will be given here. The score of candidate solution is a composite measure of two or three terms, depending on how `VIPER` is configured.

The first and primary component is the *dG_separated* term from the Rosetta application *Inter-*

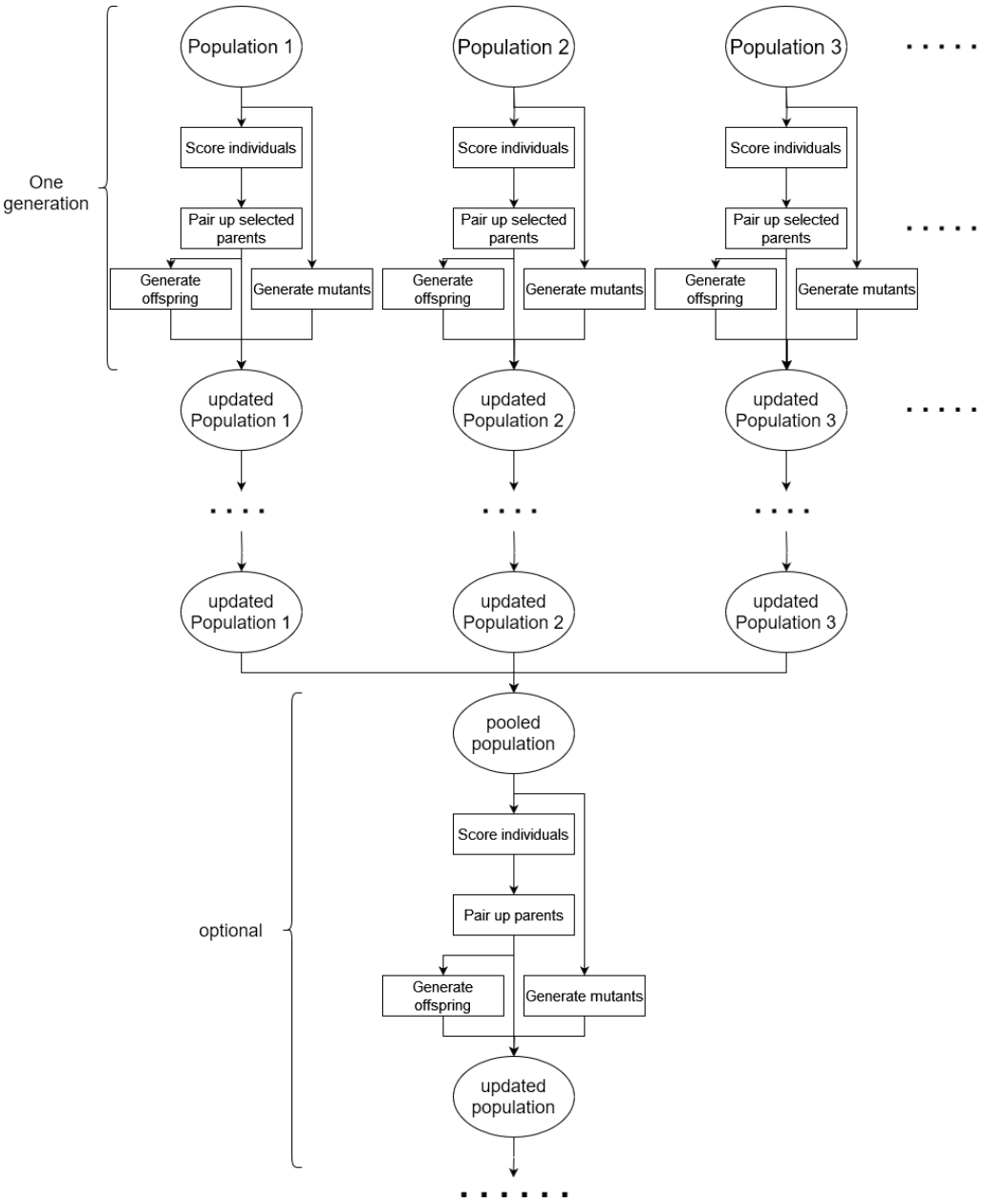


Figure 3.3: Genetic Algorithm Architecture

faceAnalyzer [1] [Lem+20]. The run configuration for this application can be found in `modules::wrappers::RosettaWrapper::Flags::interface_analyzer`. This score is a model for the binding energy and the analysis is performed on a complex of the viral surface protein and candidate peptide, superimposed on to the original binding site identified in the complex analysis step. For more information on this superimpose step, please refer to `util::PDBtool::superimpose_single`.

[1] `https://www.rosettacommons.org/docs/latest/application_documentation/analysis/interface-analyzer`

This score is then modified based on the predicted structural stability of the peptide. The prediction is performed via the spatial SCII model, a custom derivative of the side-chain interaction index proposed by Gehenn et al. [GPR03; GSR06], the code for which can be found in `util::SCII.py`. In short, the spatial SCII calculates a stability index for each residue, based on all intra-peptide sidechain interactions within a certain radius. This necessitates knowing the tertiary structure of the peptide in question, which is predicted using the PEPstrMOD webservice [KGR07; Sin+15]. More information on this procedure can be found in `modules::wrappers::PEPstrMODWrapper.py`. Based on the predicted stability, a percentage-based modifier is calculated and applied to the *dG_separated* term. The actual formula for calculating this modifier is shown in the following equation, but can be configured using the settings discussed at the end of this section.

$$\text{score}_{\text{mod}} = \text{score}_{\text{Rosetta}} \cdot \text{bonus} \tag{3.16}$$

$$\text{bonus} = \text{round}\left((\text{sSCII} - t) \cdot \frac{0.1}{w}, 1\right) \cdot 10 \cdot b + 1 \tag{3.17}$$

with

$$\text{sSCII} = \text{The spatial SCII value for the peptide in question.} \tag{3.18}$$

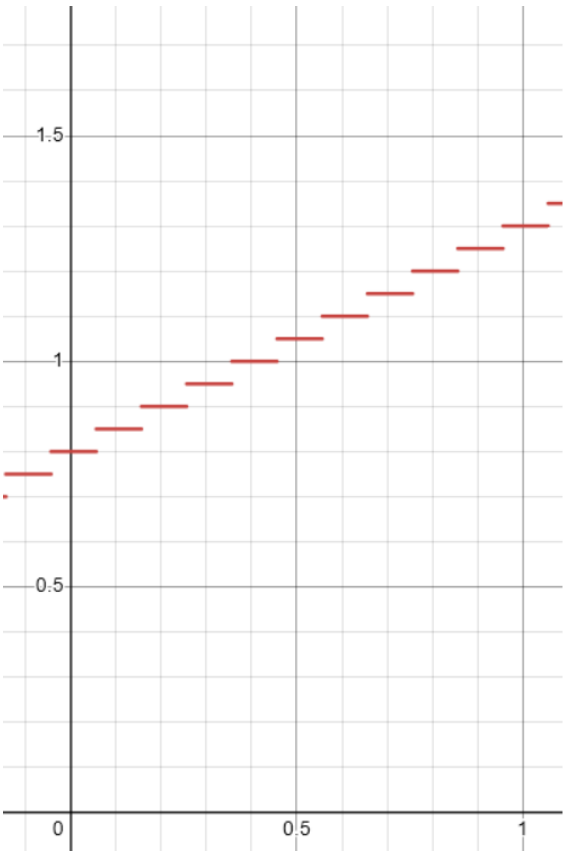
$$t = \text{The threshold between stable and unstable sSCII scores.} \tag{3.19}$$

$$w = \text{The 'width' of a bonus value step.} \tag{3.20}$$

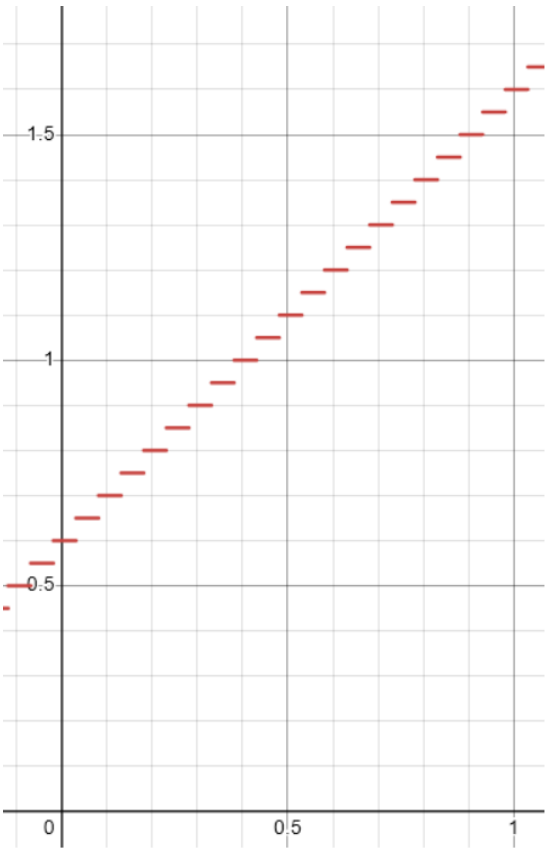
$$b = \text{The bonus value to add or deduct per step.} \tag{3.21}$$

To provide a visual intuition, figure 3.4 shows two different configurations of this function. You may replace the function that calculates this modifier with your own by implementing `custom_funcs::custom_scii_bonus` and setting `optimize.ga.scii.custom_func` to `true`. All settings for the spatial SCII calculation are described below.

Name	Type	Description	Default
		`optimize.ga.scii.`	
`adjust_score`	boolean	Whether to modify the Rosetta *dG_separated* based on spatial SCII information.	True
`radius`	float	The radius within which to consider inter-residue sidechain interactions.	7.0
`threshold`	float	The threshold for the spatial SCII value, above which a peptide should be considered stable.	0.4063
`stepping_width`	float	The 'width' of a modifier step, so for how long a peptide should have the same modifier before moving to the next highest / lowest modifier level.	0.1
`bonus`	float	By how much in percent to increase or decrease the score modifier for each step.	0.05
`custom_func`	boolean	Whether to use the custom score modification function from `custom_funcs.py`	False



(a) Plot of the sSCII-dependent bonus calculation for $t = 0.4063$, $w = 0.1$, and $b = 0.05$. Values around 0.4063 gain no bonuses, while scores below and above get progressively higher penalties or bonuses, respectively.



(b) Plot of the sSCII-dependent bonus calculation for $t = 0.4063$, $w = 0.05$, and $b = 0.05$. Values around 0.4063 gain no bonuses, while scores below and above get progressively higher penalties or bonuses, respectively.

Figure 3.4: Examples for the sSCII bonus calculation

The GA can optionally consider one more component in the scoring, which is called 'contact checking'. Essentially, `VIPER` will compare which residues in the viral surface protein the residues in the candidate peptide interact with to the residues the originally selected receptor residues interact with. Then, a configurable penalty can be applied per interaction that isn't preserved by the GA. You can set a threshold, so that a certain number of missing interactions is tolerated, as well as a range of n residues around the original target residues that will still count for the original interaction, such that interactions with neighbors may also count.

`VIPER` will always check whether there are missing contacts exceeding the threshold and write that out to a JSON file. This file (`warnings.json`) will be saved in the general folder of a GA candidate: `output/candidates/x/GA/genx_SEQUENCE/`. By default, however, no score modification is done, as losing original contacts alone isn't sufficient to point towards a worse inhibitory peptide, as new contacts could be made, or a slightly different mode of binding may be explored.

You can use the following settings to customize the contact checking.

Name	Type	Description	Default
optimize.ga.contact_checking.			
adjust_score	boolean	Whether to modify the Rosetta *dG_separated* based on contact checking information.	False
emit_warning	boolean	Whether to write the contact checking information to disk. This will also be done, if set to true, if no score modification is performed.	True
mismatch_tolerance	integer	How many original contacts may be missing, before a score penalty is applied.	2
nearby_partner_tolerance	integer	How many residues along the chain of the original residue may still count as an interaction with the original residue. So, with 1, only the direct neighbors would count.	1
penalty	float	By how much in percent to increase or decrease the score per residue exceeding the mismatch threshold.	0.02

Once all scores and modifiers are calculated and combined, the candidate and its score are saved to a central repository. Notably, the scoring is done in parallel using Python's built-in `multiprocessing` library.

3.4.3 Selection

The selection operator is highly configurable and supports multiple different selection modes. An overview of the configuration options is given below.

Name	Type	Description	Default
optimize.ga.			
select_percent	float	In percent, how many parents to select from the current population.	0.3
selection_mode	{ROULETTEWHEEL ∨ UNIFORM ∨ BESTONLY}	How to actually select the parents. ROULETTEWHEEL is a random choice, weighted by the individual's fitness, BESTONLY only selects the top $n\%$, and UNIFORM simply chooses random parents with equal chance.	ROULETTEWHEEL
selection_with_replacement	boolean	Whether to perform the parent selection with replacement.	True

3.4.4 Crossover

To generate new offspring from the parents, the crossover operator is applied. The setting `optimize.ga.crossover_mode` determines whether only a single crossover (`SINGLE`) between the sequences

should be done, or multiple (`MULTIPLE`), which is the default. You can also set the crossover chance at each residue using `optimize.ga.crossover_chance`, with the default being 0.1.

3.4.5 Mutation

Generating new sequences is done by mutating sequences, i. e. changing a random residue for another amino acid. You can set the mutation rate at each position in the amino acid sequence via `optimize.ga.mutation_rate`, with the default being 0.05. You may also bias the mutation to not be entirely random. For example, you can use a BLOSUM matrix as relative weights for amino acid substitutions to bias the mutation procedure. All available mutation biases are stored in `util/substitution_matrices`. You can choose from all BLOSUM matrices or use a uniform subtitution matrix, where all substitutions are equally common. You can also save your own matrices in this folder and have `VIPER` use them, but be sure that they follow the same format as the existing matrices. To select a specific mutation bias, adjust the `optimize.ga.mutation_bias` to one of the file names in the folder outlined above. The default is "BLOSUM62_shifted". It is important to use the shifted BLOSUM matrices, as they do not have negative entries and are therefore suitable for use as relative weights. The original matrices are only provided for reference.

You can also implement a custom mutation function, if you want to add specific mutations to the population yourself, by implementing `custom_funcs::addin_mutate` and setting `optimize.ga.custom_addin_mutate` to true. Do note that you will be handed the finalized new population, with the random mutants to pad the length at the end of the `Population`'s internal list. If you want all generations to be of the same size, you have to remove the same number of entries you want to add first.

3.4.6 Other Settings

There are a few other relevant settings that are outlined below.

Name	Type	Description	Default
		`optimize.`	
`do_optimization`	boolean	Whether to try and optimize the peptide suggested by the residue selection step.	True
		`optimize.ga.`	
`num_generation`	integer	How many generations to run the GA for.	5
`join_pops_after`	integer	After how many generations to merge the populations. A setting of -1 means to never merge the populations.	-1
`pop_size`	integer	How many individuals to have in a population.	10
`getstruc_backoff`	integer	When submitting a peptide structure prediction job to PEPstrMOD, wait 0 to n seconds before submitting the job. This is done to not overload the PEPstrMOD server and spread out the workload. A random number of seconds within the specified range is selected and waited for. If you have a large population size, please increase this number appropriately.	600

num_relax_individual	integer	When the predicted structure is received from PEPstrMOD, it first needs to be relaxed using Rosetta so that the following analysis works correctly. This setting determines how many relaxed structures to generate before selecting the one with the lowest energy.	10
dynamic_concurrent_scoring	boolean	Whether to try to dynamically determine the maximum possible of cores to allocate to every scoring job without oversubscribing the system's resources. It is recommended to stick with the default, False.	False

4 — All Settings Explained

This section will list and explain every single configuration option available in the `config.json` file.

Name	Type	Description	Default
`log_path`	String	The path where log files should be saved.	"Logs"
`verbose`	boolean	Whether to include additional information in the log file. This will *significantly* increase the size of the log file, but might make it more easy to diagnose issues.	False
`num_CPU_cores`	integer	How many CPU cores your system has. This is required if you want to use `optimize.ga.dynamic_concurrent_scoring`.	8
`results_path`	String	The path to the folder where all `VIPER` output should be saved.	"output"
`vsp_chain`	String	The chain(s) that represent the viral surface protein.	""
`partner_chain`	String	The chain(s) that represent the receptor protein.	""
`reuse_preprocessed`	boolean	Whether to reuse a relaxed PDB of the complex, if such a file can be identified.	True
`remove_other_chains`	boolean	Whether to remove all chains from the input PDB that aren't either a viral surface protein chain, or a receptor chain.	True
`rosetta_config.`			

Name	Type	Description	Default
path	String	The path to the Rosetta applications. VIPER will search the folder and all subfolders for Rosetta applications.	""
path_out	String	The output subfolder to put output from and for Rosetta applications that aren't better placed somewhere else.	"rosetta_output"
random_seed	integer	The number to seed Rosetta's random number generator with.	3333333
use_num_cores	integer	How many cores Rosetta should use.	8
archive_intermediate	boolean	Whether to archive intermediate PDB files to save storage place. *This feature isn't implemented yet!*	False
prerelax_complex_runs	integer	How many relaxed structures of the input complex to generate.	100
relax_partner_runs	integer	How many regular relaxations to run on the receptor protein in isolation. *This is currently not used!*	100
relax_peptide_nmr_runs	integer	How many runs of normal mode relaxation to run on the peptide in isolation. *This is currently not used!*	40
relax_peptide_bb_runs	integer	How many runs of backrub relaxation to run on the peptide in isolation. *This is currently not used!*	30
relax_peptide_bb_ntrials	integer	The number of Monte Carlo trials to run. *This is currently not used!*	20000
relax_peptide_fast_runs	integer	How many fast relaxations to run on the peptide in isolation. *This is currently not used!*	30
docking_runs	integer	How many docking runs to perform. *This is currently not used!*	5000
docking_translation	integer	The standard deviation of the translation to apply during docking. *This is currently not used!*	3
docking_rotation	integer	The standard deviation of the rotation to apply during docking. *This is currently not used!*	8
refine_runs	integer	How many rounds of docking refinement to run. *This is currently not used!*	100

Name	Type	Description	Default
reb_only_use_best	boolean	Whether to only use a residue energy breakdown of the lowest energy structure found during relaxation of the input complex, or aggregate the residue energy breakdown results over all generated relaxed structures.	True
gromacs_config.			
path	String	*This feature isn't implemented yet!*	""
random_seed	integer	*This feature isn't implemented yet!*	0
peptide_generator.			
use_strategy	{"fragment_joiner" ∨ "greedy_expand"}	Which residue selection strategy to use.	"fragment_joiner"
linker	String	What linker to use. Put a string of amino acids **in single letter notation** here. Only the canonical amino acids are supported.	"GG"
linker_oversize_policy	{"truncate" ∨ "ignore" ∨ "skip"}	What to do if the linker is larger than the gap it's supposed to bridge. The first option truncates the linker to exactly fit the gap, the second option simply inserts the linker as is, and the third option skips inserting a linker.	"truncate"
linking_force_length_limit	boolean	Whether to emit a warning and prematurely return during peptide assembly if inserting a linker would make the peptide length exceed the limit. The peptide will be assembled up to the limit and then terminate.	True
max_length	integer	The maximum length of the peptide to generate. This doesn't guarantee that your peptide will be of this length, but places an upper bound on the length.	18
length_damping	boolean	Whether to apply length damping, modifying the score of residues based on the number of residues that have already been included in the current selection of residues.	False
length_damping_mode	{"linear" ∨ "quadratic"}	Which type of interpolation to do between the start and end points.	"quadratic"
length_damping_min_length	integer	The length at which to start modifying the interaction energies.	2
length_damping_max_length	integer	The length at which to stop modifying the interaction energies.	7

Name	Type	Description	Default
length_ damping_ initial_mult	float	What multiplier to apply to the interaction energies up to and including the min_length.	1.0
length_ damping_ final_mult	float	What multiplied to apply to the max_length residue and all following residues. VIPER will interpolate inbetween.	0.2
length_ damping_ linear_ stepping	{float ∨ String}	By how much to increase or decrease the modifier per additional residue when linearily interpolating. Leave this as an empty String if you want to let VIPER automatically calculate the correct stepping that creates a continuous curve.	“”
custom_ strategy	boolean	Whether to use the custom residue selection strategy defined in custom_ funcs.py.	False
peptide_generator.greedy_expand.			
energy_ thresh	float	The threshold below which the aggregate interaction energy with the viral surface protein has to lie to be included in the contiguous section of residues the algorithm is currently growing.	-0.1
max_side_ extension	integer	How many neighbors on each side of the start residue to include at most.	2
ignore_ neighbors	boolean	Whether to ignore including neighbors of the start residues.	False
always_ include_ direct_ neighbors	boolean	Whether to always include the residues directly neighboring the start residue. This supersedes ignore_neighbors.	False
custom_func	boolean	Whether to use the custom residue inclusion function from custom_funcs.py	False
peptide_generator.fragment_joiner			
use_abs_ increase	boolean	Whether a residue must lower the total fragment energy in absolute terms, if it were to be included.	True
use_rel_ increase	boolean	Whether a residue must lower the total fragment energy in relative terms, if it were to be included.	False
mixed_mode_ strict	boolean	Whether a residue must fulfill both of the above criteria to be included in the current fragment.	False

Name	Type	Description	Default
`min_abs_increase`	float	By how much in absolute terms a residue must lower the total fragment energy to be included.	-0.2
`min_rel_increase`	float	By how much in relative terms (percentage) a residue must lower the total fragment energy to be included.	0.1
`lookahead`	integer	How many residues to look ahead for when encountering a residue that doesn't interact favorably with the viral surface protein in order to try to extend the current fragment anyway.	2
`linker_stretch_factor`	float	In ångström, how long the backbone of each residue in the linker is. This used to determine which stretches of residues from the receptor protein can be reasonably connected with the linker.	4.0
`old_frag_combiner`	boolean	Whether to use the old fragment combination method, which doesn't use a binary inclusion criterion when joining fragments but rather penalizes the total interaction energy of each fragment based on the distance to the fragment with the strongest interaction energy.	False
`length_flexibility`	integer	When using the old method, a flexible budget to exceed the length limit by when combining fragments.	0
`join_distance_penalty`	float	When using the old method, the distance to the most strongly interacting fragment after which to start applying a penalty.	4.0
`join_penalty_factor`	float	A percentage based penalty to a fragment's total energy per ångström distance to the most strongly interacting fragment.	0.05
`penalize_long_residues`	boolean	Whether to apply a score penalty to fragments consisting of only a single residue.	True
`lone_residue_penalty`	float	The percentage-based modifier to apply to the interaction energy of single-residue fragments.	0.5
`pad_lone_residues`	boolean	Whether to always pad fragments consisting of only a single residues using neighboring residues.	True
`lone_residue_pad_range`	integer	How many residues on each side to include when padding single-residue fragments.	1

Name	Type	Description	Default
`custom_func`	boolean	Whether to use the custom residue inclusion fragment from `custom_funcs.py`.	False
		pepstrmod_config.	
`email`	String	Which email address to use in the corresponding form field.	" "
`simulation_time`	{"100ps" ∨ "50ps"}	For how long to run the molecular dynamics simulation.	"100ps"
`environment`	{"vac" ∨ "phil" ∨ "phob"}	What environment to simulate the peptide in, vacuum, hydrophilic, or hydrophobic.	"vac"
`download_topology`	boolean	Whether to make the topology files available for download on the website. Note that this doesn't mean that `VIPER` will download those files for you!	True
`cluster_analysis`	boolean	Whether to perform cluster analysis.	False
`download_trajectory`	boolean	Whether to make the trajectory files available for download on the website. Note that this doesn't mean that `VIPER` will download those files for you!	True
`do_energy_RMS_graph`	boolean	Whether to show an energy RMS graph on the website. Note that this doesn't mean that `VIPER` will download this graph for you!	True
`wait_interval`	integer	How many seconds to wait between checking whether the results are ready. Please don't lower this value too much, so as not to do overload the webserver.	60
		`optimize.`	
`do_optimization`	boolean	Whether to try to optimize the suggested peptide.	True
		`optimize.ga.`	
`select_percent`	float	In percent, how many individuals of the population to select as parents.	0.3
`selection_mode`	{"ROULETTE-WHEEL" ∨ "BESTONLY" ∨ "UNIFORM"}	How to select parents, random but weighted by their score (fitness), only taking the *n* top percent, or completely random choice, respectively.	"ROULETTE-WHEEL"
`selection_with_replacement`	boolean	Whether to sample parents with replacement or not.	True

Name	Type	Description	Default
crossover_mode	{"SINGLE" ∨ "MULTIPLE"}	Whether to only crossover parents at a single point in the sequence or multiple.	"MULTIPLE"
crossover_chance	float	The chance at each residue to cross over into the other parent.	0.1
mutation_rate	float	The chance at each residue to mutate it into another one.	0.05
mutation_bias	see util::substitution_matrices::submat.py	What bias to apply to the mutation operator. Use "UNIFORM" if you want unbiased mutation.	"BLOSUM62_shifted"
num_generations	integer	How many generations to run.	5
join_pops_after	integer	After how many generations to merge all populations. Use -1 if you do not want to merge the populations.	-1
pop_size	integer	How many individuals should be in each population.	10
getstruc_backoff	integer	Before submitting a structure prediction job for a candidate peptide to PEPstr-MOD, a random time between 0 and getstruc_backoff is waited. Please don't set this too low in order to not stress the webserver.	600
num_relax_individual	integer	How many relaxed structures to create of each predicted peptide tertiary structure.	10
dynamic_concurrent_scoring	boolean	Whether to try to automatically determine the best way to distribute your computer's resources among scoring jobs. *This setting isn't recommended at this point!*	False
custom_addin_mutate	boolean	Whether to use the custom addin mutation functions from custom_funds.py when updating populations in a generation.	False
optimize.ga.contact_checking			
adjust_score	boolean	Whether to adjust a candidate peptide's score based on the contact checking information.	False
emit_warning	boolean	Whether to emit a warning and write it to disk if a residue exceeds the specified threshold.	True
mismatch_tolerance	integer	How many original contacts may be missing before the peptide's score will be modifier.	2

Name	Type	Description	Default
`nearby_partner_tolerance`	integer	How many residues away from the original residue along the chain may still count as an original contact with that residue.	1
`penalty`	float	A percentage-based penalty to apply to the peptide's score per residue that exceeds the mismatch tolerance.	0.02
		`optimize.ga.scii.`	
`adjust_score`	boolean	Whether to adjust the peptide's score based on the spatial SCII information.	True
`radius`	float	In ångström, how far away an amino acid may at most be to still be included in the per-residue sidechain stability index calculation.	7.0
`threshold`	float	The spatial SCII value above which a peptide should be considered stable.	0.4063
`stepping_width`	float	For how long to remain on a single modifier value before stepping up or down to the next modifier value.	0.1
`bonus`	float	By how much to increase or decrease the modifier multiplier per step.	0.05
`custom_func`	boolean	Whether to use the custom spatial SCII modifier derivation function from `custom_funcs.py`.	False

5 — Modifying VIPER

If you would like to go further the customization options provided via the `config.json` file, or even extend `VIPER`, this section will provide some additional background information about how best to do this.

5.1 Program Control Flow

The nexus regarding the `VIPER` program flow is the `VIPER.py` file. If you want to largely divert the the program flow or insert your own modules, this is the place to go. You can slot your module into `VIPER::VIPER::run` method at the appropriate place. This class bundles the highest level abstract steps that need to be taken and can serve as a natural entry point for large scale modifications.

5.2 Program Structure

`VIPER` makes use of modules as a layer of abstraction. This means that while the VIPER class outlined above bundles high level, abstract steps, such as "optimize this peptide", the actual logic implementing this functionality in a separate layer. To do so, modules are conceptually split into wrappers and stages. The former provides a way to interface with external software tools, while the latter is supposed to hold mostly self-contained logic. If you are intent on adding additional utility or convenience functions, you can consider placing them in `util`. Lastly, `modules::interfaces` provides the base templates for three types of modules: `OptimizationStrategy`, which provides the interface to standardize the communication between a module that can optimize a peptide and the rest of `VIPER`, `ResSelectionStrategy`, which provides the interface to standardize the communication between a module that provides an initial candidate solution and the rest of `VIPER`, and `StructureProvider`, which provides the interface to standardize the communciation between a module that can provide the tertiary structure for a protein/peptide and the rest of `VIPER`. If you implement a new module that implements any of these interfaces, you will find it easier to swap out the usage of an existing module with your module. For example, if you want to implement a module that uses a machine learning solution to predict the tertiary structure of a peptide, you can

simply implement `StructureProvider` and swap out the usage of the current StructureProvider (`PEPstrMODWrapper`) for your module.

5.3 RosettaWrapper

`modules::wrappers::RosettaWrapper::RosettaWrapper` provides functionality for running arbitrary Rosetta applications with arbitrary configuration options. To do so, have a look at the `run` function, which takes as the first argument a dictionary of application configuration options, which will be written to a flag file and then be used for the application specified in the "app" field of the dictionary. `RosettaWrapper` provides a few preconfigured flag dictionaries in the `RosettaWrapper::Flags` class, which you can easily reuse and adapt to fit your use case. `RosettaWrapper` will also fill in the random seed information, if the corresponding flags exist as a key in the dictionary.

Bibliography

[GPR03] Katja Gehenn, Rüdiger Pipkorn, and Jennifer Reed. "Successful Design and Synthesis of a Polarity-Triggered beta → alpha Conformational Switch Using the Side Chain Interaction Index (SCII) as a Measure of Local Stuctural Stability". In: *Biochemistry* 43.3 (Dec. 2003), pages 607–612. ISSN: 1520-4995. DOI: 10.1021/bi0301744. URL: http://dx.doi.org/10.1021/bi0301744 (cited on page 23).

[GSR06] Katja Gehenn, Julia Stege, and Jennifer Reed. "The side chain interaction index as a tool for predicting fast-folding elements and the structure and stability of engineered peptides". In: *Analytical Biochemistry* 356.1 (Sept. 2006), pages 12–17. ISSN: 0003-2697. DOI: 10.1016/j.ab.2006.06.021. URL: http://dx.doi.org/10.1016/j.ab.2006.06.021 (cited on page 23).

[KGR07] Harpreet Kaur, Aarti Garg, and G.P.S. Raghava. "PEPstr: A de novo Method for Tertiary Structure Prediction of Small Bioactive Peptides". In: *Protein & Peptide Letters* 14.7 (July 2007), pages 626–631. ISSN: 0929-8665. DOI: 10.2174/092986607781483859. URL: http://dx.doi.org/10.2174/092986607781483859 (cited on pages 6, 23).

[Kha+11] Firas Khatib et al. "Algorithm discovery by protein folding game players". In: *Proceedings of the National Academy of Sciences* 108.47 (Nov. 2011), pages 18949–18953. ISSN: 1091-6490. DOI: 10.1073/pnas.1115898108. URL: http://dx.doi.org/10.1073/pnas.1115898108 (cited on pages 6, 11).

[Lem+20] Julia Koehler Leman et al. "Macromolecular modeling and design in Rosetta: recent methods and frameworks". In: *Nature Methods* 17.7 (June 2020), pages 665–680. ISSN: 1548-7105. DOI: 10.1038/s41592-020-0848-2. URL: http://dx.doi.org/10.1038/s41592-020-0848-2 (cited on pages 6, 12, 22).

[Mag+20] Jack B. Maguire et al. "Perturbing the energy landscape for improved packing during computational protein design". In: *Proteins: Structure, Function, and Bioinformatics* 89.4 (Dec. 2020), pages 436–449. ISSN: 1097-0134. DOI: 10.1002/prot.26030. URL: http://dx.doi.org/10.1002/prot.26030 (cited on pages 6, 11).

[Sin+15] Sandeep Singh et al. “PEPstrMOD: structure prediction of peptides containing natural, non-natural and modified residues”. In: *Biology Direct* 10.1 (Dec. 2015). ISSN: 1745-6150. DOI: 10.1186/s13062-015-0103-4. URL: http://dx.doi.org/10.1186/s13062-015-0103-4 (cited on pages 6, 23).