Opfi

Alexis M. Hill & Jim Rybarski

Oct 26, 2021

CONTENTS

Welcome to the Opfi documentation site! Opfi is a modular, rule-based framework for creating gene cluster identification pipelines, particularly for large genomics or metagenomics datasets.

Opfi is implemented entirely in Python, and can be downloaded with conda or the from the Python Package Index. It consists of two major modules: Gene Finder, for discovery of novel gene clusters, and Operon Analyzer, for rule-based filtering, deduplication, visualization, and re-annotation of systems identified by Gene Finder.

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Installation

The recommended way to install Opfi is with Bioconda, which requires the conda package manager. This will install Opfi and all of its dependencies (which you can read more about below, see *Dependencies*).

Currently, Bioconda supports only 64-bit Linux and Mac OS. Windows users can still install Opfi with pip (see below); however, the complete installation procedure has not been fully tested on a Windows system.

Install with conda (Linux and Mac OS only)

First, set up conda and Bioconda following the quickstart guide. Once this is done, run:

```
conda install -c bioconda opfi
```

And that's it! Note that this will install Opfi in the conda environment that is currently active. To create a fresh environment with Opfi installed, do:

```
conda create --name opfi-env -c bioconda opfi
conda activate opfi-env
```

Install with pip

This method does not automatically install non-Python dependencies, so they will need to be installed separately, following their individual installation instructions. A complete list of required software is provided below, see *Dependencies*. Once this step is complete, install Opfi with pip by running:

```
pip install opfi
```

Install from source

Finally, the latest development build may be installed directly from Github. First, non-Python *Dependencies* will need to be installed in the working environment. An easy way to do this is to first install Opfi with conda using the *Install with conda (Linux and Mac OS only)* method (we'll re-install the development version of the Opfi package in the next step). Alternatively, dependencies can be installed individually.

Once dependencies have been installed in the working environment, run the following code to download and install the development build:

```
git clone https://github.com/wilkelab/Opfi.git
cd Opfi
pip install . # or pip install -e . for an editable version
pip install -r requirements # if conda was used, this can be skipped
```

Testing the build

Regardless of installation method, users can download and run Opfi's suite of unit tests to confirm that the build is working as expected. First download the tests from Github:

```
git clone https://github.com/wilkelab/Opfi
cd Opfi
```

And then run the test suite using pytest:

```
pytest --runslow --runmmseqs --rundiamond
```

This may take a minute or so to complete.

1.1.2 Dependencies

Opfi uses the following bioinformatics software packages to find and annotate genomic features:

Table 1.1: Software dependencies

Application	Description
NCBI BLAST+	Protein and nucleic acid homology search tool
Diamond	Alternative to BLAST+ for fast protein homology searches
MMseqs2	Alternative to BLAST+ for fast protein homology searches
PILER-CR	CRISPR repeat detection
Generic Repeat Finder	Transposon-associated repeat detection

The first three (BLAST+, Diamond, and MMseqs2) are popular homology search applications, that is, programs that look for local similarities between input sequences (either protein or nucleic acid) and a target. These are used by Opfi in *gene_finder.pipeline.Pipeline* for annotation of genes or non-coding regions of interest in the input genome/contig. The user specifies which homology search tool to use during pipeline setup (see *gene_finder.pipeline.Pipeline* for details). Note that the BLAST+ distribution contains multiple programs for homology searching, three of which (blastp, blastn, and PSI-BLAST) are currently supported by Opfi.

The following table summarizes the main difference between each homology search program. It may help users decide which application will best meet their needs. Note that performance tests are inherently hardware and context dependent, so this should be taken as a loose guide, rather than a definitive comparison.

Table 1.2: Comparison of homology search programs supported by Opfi

Application	Relative sensitivity	Relative speed	Requires a protein or nucleic acid sequence database?
Diamond	+	++++	protein
MMseqs2	++	+++	protein
blastp	+++	++	protein
PSI-BLAST	++++	+	protein
blastn	NA	NA	nucleic acid

The last two software dependencies, PILER-CR and Generic Repeat Finder (GRF), deal with annotation of repetive sequences in DNA. PILER-CR identifies CRISPR arrays, regions of alternatating ~30 bp direct repeat and variable sequences that play a role in prokaryotic immunity. GRF identifies repeats associated with transposable elements, such as terminal inverted repeats (TIRs) and long terminal repeats (LTRs).

1.2 Example Usage

1.2.1 Example 1: Finding CRISPR-Cas systems in a cyanobacteria genome

In this example, we will annotate and visualize CRISPR-Cas systems in the cyanobacteria species Rippkaea orientalis. CRISPR-Cas is a widespread bacterial defense system, found in at least 50% of all known prokaryotic species. This system is significant in that it can be leveraged as a precision gene editing tool, an advancement that was awarded the 2020 Nobel Prize in Chemistry. The genome of R. orientalis harbors two complete CRISPR-Cas loci (one chromosomal, and one extrachromosomal/plasmid).

You can download the complete assembled genome here; it is also available at https://github.com/wilkelab/Opfi under `tutorials`, along with the other data files necessary to run these examples, and an interactive jupyter notebook version of this tutorial.

This tutorial assumes the user has already installed Opfi and all dependencies (if installing with conda, this is done automatically). Some familiarity with BLAST and the basic homology search algorithm may also be helpful, but is not required.

1. Use the makeblastdb utility to convert a Cas protein database to BLAST format

We start by converting a Cas sequence database to a format that BLAST can recognize, using the command line utility **`makeblastdb`**, which is part of the core NCBI BLAST+ distribution. A set of ~20,000 non-redundant Cas sequences, downloaded from Uniprot is available as a tar archive `tutorials/cas_database.tar.gz` . We'll make a new directory, "blastdb", and extract sequences there:

```
mkdir blastdb
cd blastdb && tar -xzf cas_database.tar.gz && cd ..
```

Next, create two BLAST databases for the sequence data: one containing Cas1 sequences only, and another that contains the remaining Cas sequences.

```
cd blastdb && cat cas1.fasta | makeblastdb -dbtype prot -title cas1 -hash_index -out␣
↪cas1_db && cd ..
cd blastdb && cat cas[2-9].fasta cas1[0-2].fasta casphi.fasta | makeblastdb -dbtype prot␣
↪-title cas_all -hash_index -out cas_all_but_1_db && cd ..
```

`-dbtype prot` simply tells **makeblastdb** to expect amino acid sequences. We use `-title` and `-out` to name the database (required by BLAST) and to prefix the database files, respectively. `-hash_index` directs **makeblastdb** to generate a hash index of protein sequences, which can speed up computation time.

2. Use Gene Finder to search for CRISPR-Cas loci

CRISPR-Cas systems are extremely diverse. The most recent classification effort identifies 6 major types, and over 40 subtypes, of compositionally destinct systems. Although there is sufficent sequence similarity between subtypes to infer the existence of a common ancestor, the only protein family present in the majority of CRISPR-cas subtypes is the conserved endonuclease Cas1. For our search, we will define candidate CRISPR-cas loci as having, minimally, a cas1 gene.

First, create another directory for output:

```
mkdir example_1_output
```

The following bit of code uses Opfi's *gene_finder.pipeline* module to search for CRISPR-Cas systems:

```
from gene_finder.pipeline import Pipeline
import os

genomic_data = "GCF_000024045.1_ASM2404v1_genomic.fna.gz"
output_directory = "example_1_output"

p = Pipeline()
p.add_seed_step(db="blastdb/cas1_db", name="cas1", e_val=0.001, blast_type="PROT", num_
↪threads=1)
p.add_filter_step(db="blastdb/cas_all_but_1_db", name="cas_all", e_val=0.001, blast_type=
↪"PROT", num_threads=1)
p.add_crispr_step()

# use the input filename as the job id
# results will be written to the file <job id>_results.csv
job_id = os.path.basename(genomic_data)
results = p.run(job_id=job_id, data=genomic_data, output_directory=output_directory, min_
↪prot_len=90, span=10000, gzip=True)
```

First, we initialize a *gene_finder.pipeline.Pipeline* object, which keeps track of all search parameters, as well as a running list of systems that meet search criteria. Next, we add three search steps to the pipeline:

1. *gene_finder.pipeline.Pipeline.add_seed_step()* : BLAST is used to search the input genome against a database of Cas1 sequences. Regions around putative Cas1 hits become the intial candidates, and the rest of the genome is ignored.
2. *gene_finder.pipeline.Pipeline.add_filter_step()* : Candidate regions are searched for any additional Cas genes. Candidates without at least one additional putative Cas gene are also discarded.
3. *gene_finder.pipeline.Pipeline.add_crispr_step()* : Remaining candidates are annotated for CRISPR repeat sequences using PILER-CR.

Finally, we run the pipeline, executing steps in the order they we added. `min_prot_len` sets the minimum length (in amino acid residues) of hits to keep (really short hits are unlikely real protein encoding genes). `span` is the region directly up- and downstream of initial hits. So, each candidate system will be about 20 kbp in length. Results are written to a single CSV file. Final candidate loci contain at least one putative Cas1 gene and one additional Cas gene. As we will see, this relatively permissive criteria captures some non-CRISPR-Cas loci. Opfi has additional modules for reducing unlikely systems after the gene finding stage.

3. Visualize annotated CRISPR-Cas gene clusters with Operon Analyzer

It is sometimes useful to visualize candidate systems, especially during the exploratory phase of a genomics survey. Opfi provides a few functions for visualizing candidate systems in *operon_analyzer.visualize*. We'll use these to visualize the CRISPR-Cas gene clusters in R. orientalis:

```
import csv
import sys
from operon_analyzer import load, visualize

feature_colors = { "cas1": "lightblue",
                   "cas2": "seagreen",
                   "cas3": "gold",
                   "cas4": "springgreen",
                   "cas5": "darkred",
                   "cas6": "thistle",
                   "cas7": "coral",
                   "cas8": "red",
                   "cas9": "palegreen",
                   "cas10": "yellow",
                   "cas11": "tan",
                   "cas12": "orange",
                   "cas13": "saddlebrown",
                   "casphi": "olive",
                   "CRISPR array": "purple"
                   }

# read in the output from Gene Finder and create a gene diagram for each cluster (operon)
with open("example_1_output/GCF_000024045.1_ASM2404v1_genomic.fna.gz_results.csv", "r")␣
↪as operon_data:
    operons = load.load_operons(operon_data)
    visualize.plot_operons(operons=operons, output_directory="example_1_output", feature_
↪colors=feature_colors, nucl_per_line=25000)
```

Running this script produces the following three gene diagrams, one for each system in the input CSV:

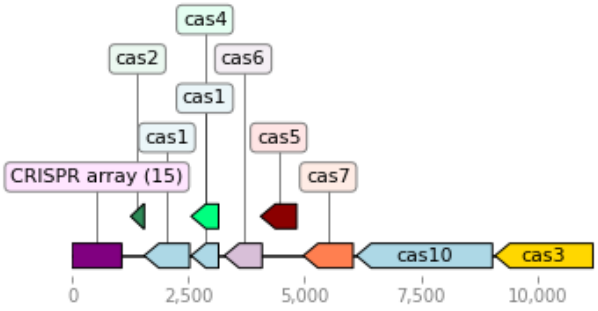

Fig. 1.1: A CRISPR-Cas system in the chromosome of R. orientalis.

We can see that both CRISPR-Cas systems were identified (Fig. 1.1 and Fig. 1.2). We also see some systems that don't resemble functional CRISPR-Cas operons (Fig. 1.3). Because we used a relatively permissive e-value threshhold

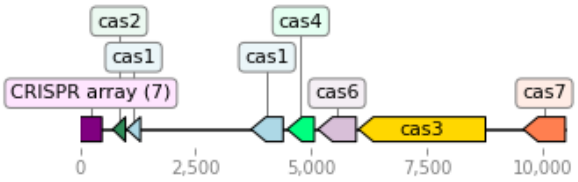

Fig. 1.2: A second CRISPR-Cas system in R. orientalis plasmid 1.

Fig. 1.3: An R. orientalis locus with a putative CRISPR-Cas gene.

of 0.001 when running BLAST, Opfi retained regions with very low sequence similarity to true CRISPR-Cas genes. In fact, these regions are likely not CRISPR-Cas loci at all. Using a lower e-value would likely eliminate these "false positive" systems, but `operon_analyzer.rules` exposes functions for filtering out unlikely candidates *after* the intial BLAST search.

In general, we have found that using permissive BLAST parameters intially, and then filtering or eliminating candidates during the downstream analysis, is an effective way to search for gene clusters in large amounts of genomic/metagenomic data. In this toy example, we could re-run BLAST many times without significant cost. But on a more realistic dataset, needing to re-do the computationally expensive homology search could detrail a project. Since the optimal search parameters may not be known *a priori*, it can be better to do a permissive homology search initially, and then narrow down results later.

Finally, clean up the temporary directories, if desired:

```
rm -r example_1_output blastdb
```

1.2.2 Example 2: Filter and classify CRISPR-Cas systems based on genomic composition

As discussed in the previous example, known CRISPR-Cas systems fall into 6 broad categories, based on the presence of particular "signature" genes, as well as overall composition and genomic architecture. In this example, we will use Opfi to search for and classify CRISPR-Cas systems in ~300 strains of fusobacteria.

This dataset was chosen because it is more representative (in magnitude) of what would be encountered in a real genomics study. Additionally, the fusobacteria phylum contains a variety of CRISPR-Cas subtypes. Given that the homology search portion of the analysis takes several hours (using a single core) to complete, we have pre-run Gene Finder using the same setup as the previous example.

1. Make another temporary directory for output:

```
mkdir example_2_output
```

2. Filter Gene Finder output and extract high-confidence CRISPR-Cas systems

The following code reads in unfiltered output from *gene_finder.pipeline.Pipeline* and applies a set of conditions ("rules") to accomplish two things: 1. Select (and bin) systems according to type, and, 2. Eliminate candidates that likely do not represent true CRISPR-Cas systems

To do this, we'll leverage the *operon_analyzer.rules* and *operon_analyzer.analyze* modules.

```
from operon_analyzer import analyze, rules


fs = rules.FilterSet().pick_overlapping_features_by_bit_score(0.9)
cas_types = ["I", "II", "III", "V"]

rulesets = []
# type I rules
rulesets.append(rules.RuleSet().contains_group(feature_names = ["cas5", "cas7"], max_gap_
↪distance_bp = 1000, require_same_orientation = True) \
                               .require("cas3"))
# type II rules
rulesets.append(rules.RuleSet().contains_at_least_n_features(feature_names = ["cas1",
↪"cas2", "cas9"], feature_count = 3) \
                               .minimum_size("cas9", 3000))
# type III rules
rulesets.append(rules.RuleSet().contains_group(feature_names = ["cas5", "cas7"], max_gap_
↪distance_bp = 1000, require_same_orientation = True) \
                               .require("cas10"))
# type V rules
rulesets.append(rules.RuleSet().contains_at_least_n_features(feature_names = ["cas1",
↪"cas2", "cas12"], feature_count = 3))

for rs, cas_type in zip(rulesets, cas_types):
    with open("refseq_fusobacteria.csv", "r") as input_csv:
        with open(f"example_2_output/refseq_fuso_filtered_type{cas_type}.csv", "w") as␣
↪output_csv:
            analyze.evaluate_rules_and_reserialize(input_csv, rs, fs, output_csv)
```

The rule sets are informed by an established CRISPR-Cas classification system, which you can learn more about in this paper . The most recent system recognizes 6 major CRISPR-Cas types, but since fusobacteria doesn't contain type IV or VI systems that can be identified with our protein dataset, we didn't define the corresponding rule sets.

3. Verify results with additional visualizations

Altogther, this analysis will identify several hundred systems. We won't look at each system individually (but you are free to do so!). For the sake of confirming that the code ran as expected, we'll create gene diagrams for just the type V systems, since there are only two:

```
import csv
import sys
from operon_analyzer import load, visualize

feature_colors = { "cas1": "lightblue",
                   "cas2": "seagreen",
                   "cas3": "gold",
                   "cas4": "springgreen",
                   "cas5": "darkred",
                   "cas6": "thistle",
                   "cas7": "coral",
                   "cas8": "red",
                   "cas9": "palegreen",
                   "cas10": "yellow",
                   "cas11": "tan",
                   "cas12": "orange",
                   "cas13": "saddlebrown",
                   "casphi": "olive",
                   "CRISPR array": "purple"
                   }

# read in the output from Gene Finder and create a gene diagram for each cluster (operon)
with open("example_2_output/refseq_fuso_filtered_typeV.csv", "r") as operon_data:
    operons = load.load_operons(operon_data)
    visualize.plot_operons(operons=operons, output_directory="example_2_output", feature_
↪colors=feature_colors, nucl_per_line=25000)
```

The plotted systems should look like this:

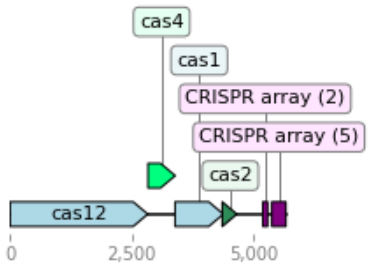

Fig. 1.4: A type V CRISPR-Cas system.

Finally, clean up the temporary output directory, if desired:

```
rm -r example_2_output
```

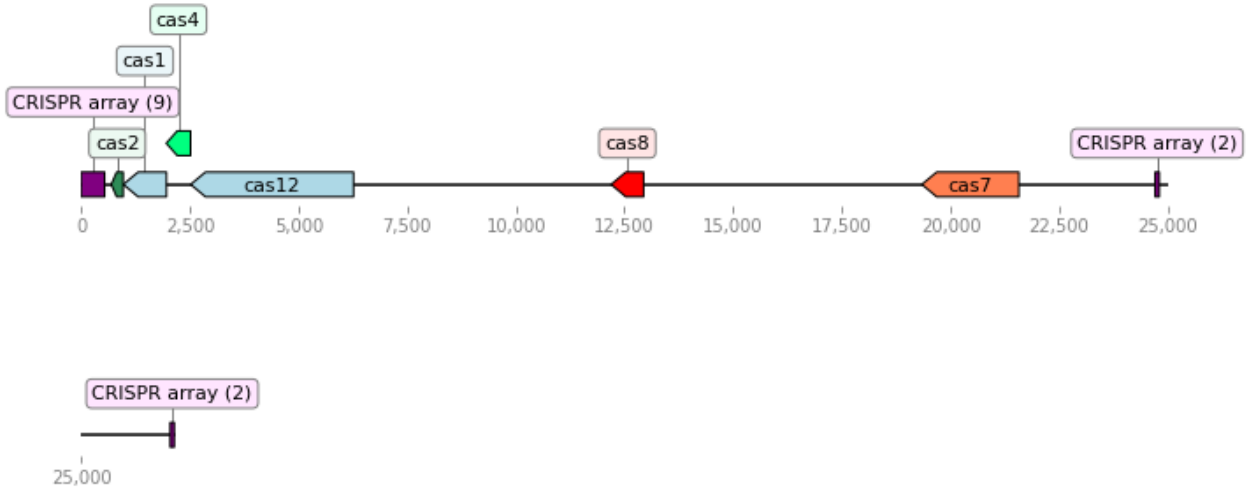

Fig. 1.5: A second type V CRISPR-Cas system.

1.3 Inputs and Outputs

1.3.1 Building sequence databases

To search for gene clusters with Opfi, users must compile representative protein (or nucleic acid) sequences for any genes expected in target clusters (or for any non-essential accessory genes of interest). These may be from a pre-existing, private collection of sequences (perhaps from a previous bioinformatics analysis). Alternatively, users may download sequences from a publically available database such as Uniprot (maintained by the European Bioinformatics Institute) or one of the databases provided by the National Center for Biotechnology Information.

Once target sequences have been compiled, they must be converted to an application-specific database format. Opfi currently supports **BLAST+**, **mmseqs2**, and **diamond** for homology searching:

- Instructions for creating sequence databases for BLAST using makeblastdb
- Instructions for creating sequence databases for mmseqs2 using mmseqs createdb
- Diamond makedb command options

The FASTA file format

Both genomic input data and reference sequence data should be in FASTA format. This is a simple flat text representation of biological sequence data, where individual sequences are delineated by the > greater than character. For example:

```
>UniRef50_Q02ML7 CRISPR-associated endonuclease Cas1 n=1700 RepID=CAS1_PSEAB
MDDISPSELKTILHSKRANLYYLQHCRVLVNGGRVEYVTDEGRHSHYWNIPIANTTSLLL
GTGTSITQAAMRELARAGVLVGFCGGGGTPLFSANEVDVEVSWLTPQSEYRPTEYLQRWV
GFWFDEEKRLVAARHFQRARLERIRHSWLEDRVLRDAGFAVDATALAVAVEDSARALEQA
PNHEHLLTEEARLSKRLFKLAAQATRYGEFVRAKRGSGGDPANRFLDHGNYLAYGLAATA
TWVLGIPHGLAVLHGKTRRGGLVFDVADLIKDSLILPQAFLSAMRGDEEQDFRQACLDNL
SRAQALDFMIDTLKDVAQRSTVSA
```

(continues on next page)

(continued from previous page)

```
>UniRef50_Q2RY21 CRISPR-associated endonuclease Cas1 1 n=1034 RepID=CAS1A_RHORT
MADPAFVPLRPIAIKDRSSIVFLQRGQLDVVDGAFVLIDQEGVRVQIPVGGLACLMLEPG
TRITHAAIVLCARVGCLVIWVGERGTRLYAAGQPGGARADRLLFQARNALDETARLNVVR
EMYRRRFDDDPPARRSVDQLRGMEGVRVREIYRLLAKKYAVDWNARRYDHNDWDGADIPN
RCLSAATACLYGLCEAAILAAGYAPAIGFLHRGKPQSFVYDVADLYKVETVVPTAFSIAA
KIAAGKGDDSPPERQVRIACRDQFRKSGLLEKIIPDIEEILRAGGLEPPLDAPEAVDPVI
PPEEPSGDDGHRG
```

The sequence definition (defline) comes directly after the > character, and should be on a separate line from the sequence (which can be on one or more subsequent lines). There is no specific defline format, however, Opfi requires that, for both genomic input and sequence data, each definition line contain a unique sequence identifer. This should be a single word/token immediately following the > character (i.e. spaces between the > character and the identifier are not allowed). Any additional text on the defline is parsed as a single string, and appears in the output CSV (see *Opfi output format*).

Tip: Biological sequences downloaded from most public databases will have an accession number/identifier by default.

Annotating sequence databases

To take full advantage of the rule-based filtering methods in *operon_analyzer.rules*, users are encouraged to annotate reference sequences with a name/label that is easily searched. Labels can be as broad or as specific as is necessary to provide meaningful annotation of target gene clusters.

Gene labels are parsed from sequence deflines; specifically, Opfi looks for the second word/token following the > character. For example, the following FASTA sequence has been annotated with the label "cas1":

```
>UniRef50_Q02ML7 cas1 CRISPR-associated endonuclease Cas1 n=1700 RepID=CAS1_PSEAB
MDDISPSELKTILHSKRANLYYLQHCRVLVNGGRVEYVTDEGRHSHYWNIPIANTTSLLL
GTGTSITQAAMRELARAGVLVGFCGGGGTPLFSANEVDVEVSWLTPQSEYRPTEYLQRWV
GFWFDEEKRLVAARHFQRARLERIRHSWLEDRVLRDAGFAVDATALAVAVEDSARALEQA
PNHEHLLTEEARLSKRLFKLAAQATRYGEFVRAKRGSGGDPANRFLDHGNYLAYGLAATA
TWVLGIPHGLAVLHGKTRRGGLVFDVADLIKDSLILPQAFLSAMRGDEEQDFRQACLDNL
SRAQALDFMIDTLKDVAQRSTVSA
```

After running *gene_finder.pipeline.Pipeline*, users could select candidates with hits against this sequence using the following rule set:

```
from operon_analyzer.rules import RuleSet

rs = RuleSet.require("cas1")
```

In practice, a genomics search might use a reference database of hundreds (or even thousands) of representative protein sequences, in which case labeling each sequence individually would be tedious. It is recommended to organize sequences into groups of related proteins that can be given a single label. This script uses the Python package **Biopython** to annotate sequences in a multi-sequence FASTA file:

```
from Bio import SeqIO
import os, sys

def annotate_reference(prot_ref_file, label):
    records = list(SeqIO.parse(ref_fasta, "fasta"))
```

(continues on next page)

(continued from previous page)

```
    for record in records:
        des = record.description.split()
        prot_id = des.pop(0)
        des_with_label = "{} {} {}".format(prot_id, label, " ".join(des))
        record.description = des_with_label

    SeqIO.write(records, ref_fasta, "fasta")

if __name__ == "__main__":
    ref_fasta = sys.argv[1]
    label = sys.argv[2]
    annotate_reference(ref_fasta, label)
```

It is possible to use the entire sequence description (i.e. all text following the sequence identifier) as the gene label. This is particularly useful when using a pre-built database like nr, which contains representative protein sequences for many different protein families. When using sequence databases that haven't been annotated, users should set `parse_descriptions=False` for each *gene_finder.pipeline.Pipeline* `add_step()` method call.

Converting sequence files to a sequence database

Once reference sequences have been compiled (and, optionally, labeled) they must be converted to a sequence database format that is specific to the homology search program used. Currently, Opfi supports **BLAST**, **mmseqs2**, and **diamond**. Each software package is automatically installed with a companion utility program for generating sequence databases. The following example shows what a typical call to **makeblastdb**, the BLAST+ database utility program, might look like:

```
makeblastdb -in "my_sequences.fasta" -out my_sequences/db -dbtype prot -title "my_
↪sequences" -hash_index
```

The command takes a text/FASTA file `my_sequences.fasta` as input, and writes the resulting database files to the directory `my_sequences`. Database files are prefixed with "db". `-dbtype prot` specifies that the input is amino acid sequences. We use `-title` to name the database (required by BLAST). `-hash_index` directs **makeblastdb** to generate a hash index of protein sequences, which can speed up computation time.

Tip: **mmseqs2** and **diamond** have similar database creation commands, see *Building sequence databases*.

1.3.2 BLAST advanced options

BLAST+ programs have a number of tunable parameters that can, for example, be used to adjust the sensitivity of the search algorithm. We anticipate that application defaults will be sufficient for most users; nevertheless, it is possible to use non-default program options by passing them as keyword arguments to *gene_finder.pipeline.Pipeline* `add_step()` methods.

For example, when using **blastp** on the command line, we could adjust the number of CPUs to four by passing the argument `-num_threads 4` to the program. When using Opfi, this would look like `num_threads=4`.

Flags (boolean arguments that generally do not precede additional data) are also possible. For example, the command line flag `-use_sw_tback` tells **blastp** to compute locally optimal Smith-Waterman alignments. The correct way to specify this behavior via the *gene_finder.pipeline.Pipeline* API would be to use the argument `use_sw_tback=True`.

Below is a list of options accepted by Opfi. Note that some BLAST+ options are not allowed, mainly those that modify BLAST output.

Program	Allowed Options
blastp and **psiblast**	dbsize word_size gapopen gapextend qcov_hsp_perc xdrop_ungap xdrop_gap xdrop_gap_final searchsp sum_stats seg soft_masking matrix threshold culling_limit window_size num_threads comp_based_stats gilist seqidlist negative_gilistdb_soft_mask db_hard_mask entrez_query max_hspsbest_hit_overhang best_hit_score_edge max_target_seqsimport_search_strategy export_search_strategy num_alignments
blastp only	mask
psiblast only	gap_trigger num_iterations out_pssm out_ascii_pssm pseudocount inclusion_ethresh
blastp (flags)	lcase_masking ungapped use_sw_tback remote
psiblast (flags)	lcase_masking use_sw_tback save_pssm_after_last_round save_each_pssm remote
blastn	filtering_algorithm sum_stats window_masker_db window_size template_type version parse_deflines min_raw_gapped_score string format max_hsps taxids negative_taxids num_alignments strand off_diagonal_range subject_besthit num_sequences no_greedy negative_taxidlist culling_limit xdrop_ungap open_penalty DUST_options sorthits xdrop_gap_final negative_gilist subject use_index bool_value filename seqidlist task_name sort_hits database_name lcase_masking query_loc subject_loc sort_hsps line_length boolean db_hard_mask negative_seqidlist template_length filtering_db filtering_database penalty searchsp ungapped type gapextend db_soft_mask dbsize qcov_hsp_perc sorthsps window_masker_taxid index_name export_search_strategy float_value soft_masking gilist entrez_query show_gis best_hit_score_edge gapopen subject_input_file range html word_size best_hit_overhang perc_identity input_file num_descriptions xdrop_gap dust taxidlist max_target_seqs num_threads task remote int_value extend_penalty reward import_search_strategy num_letters

You can read more about BLAST+ options in the BLAST+ appendices.

Note: Using advanced options with **mmseqs2** and **diamond** is not supported at this time.

1.3.3 Opfi output format

Results from *gene_finder.pipeline.Pipeline* searches are written to a single CSV file. Below is an example from the tutorial (see *Example Usage*):

NC_0[illegible] 1	07	cas1 CRISPR-associated endoribonuclease Cas2 n=2 Tax=Thermosulfurimonas dismutans TaxID=999894 RepID=A0A179D3U4_9BACT	[illegible] FYLI- [illegible] PATK- AGNKR- RKR- LYEML- CGYGNWTQFSVFECFLTAVQ- FAN- LQSKLENLIQPNEDSVRIYILDAGSVRKTLTYGSEKPRQVDTLIL	[illegible]A0A179D3U4	43.13[illegible]	[illegible]72	29	31	0	0	60.78	53	data/GCF_000024045
NC_0[illegible] 1	177	cas1 CRISPR-associated endonuclease Cas1 n=83 Tax=Cyanobacteria TaxID=1117 RepID=A0A1Z3HN48_9CYAN	[illegible] LYLTQP- [illegible]VL- [illegible]KQEAFH- VALKQEDGSWKKQLI- PAQTVE- QIVLI- GYP- SIT- GEAL- [illegible]HYLSCFGKYLGSALPGYSRNGQLRLAQYHVHDNEEQRLALVK	[illegible]A0A1Z3HN48	69.53[illegible]	[illegible]26	99	276	0	0	84.92	100	data/GCF_000024045
NC_0[illegible] 1	100	cas1 CRISPR-associated exonuclease Cas4 n=83 Tax=Cyanobacteria TaxID=1117 RepID=A0A2I8A541_9NOSO	[illegible] NYLPLAYL- [illegible]F- EYC- TR- RFY- WEYVL- GE- MAN- NEHI- [illegible] [illegible]RIHLHRN- [illegible] QEGI- IKEEDTI- IHRQQWVWSDRLQIKGIIDAVEEKESSLVPVEYKKGRMSQHLNDHFQLCAAAL	[illegible]A0A2I8A541	66.15[illegible]	[illegible]29	66	162	0	0	83.08	99	data/GCF_000024045
NC_0[illegible] 1	46	cas2 CRISPR-associated endoribonuclease Cas2 n=68 Tax=Cyanobacteria TaxID=1117 RepID=A0A1Z3HN55_9CYAN	[illegible] FYLI- [illegible] PATK- AGNKR- RKR- LYEML- CGYGNWTQFSVFECFLTAVQ- FAN- LQSKLENLIQPNEDSVRIYILDAGSVRKTLTYGSEKPRQVDTLIL	[illegible]A0A1Z3HN55	67.02[illegible]	[illegible]63	31	77	0	0	81.91	97	data/GCF_000024045

The first two columns contain the input genome/contig sequence ID (sometimes called an accession number) and the coordinates of the candidate gene cluster, respectively. Since an input file can have multiple genomic sequences, these two fields together uniquely specify a candidate gene cluster. Each row represents a single annotated feature in the candidate locus. Features from the same candidate are always grouped together in the CSV.

Descriptions of each output field are provided below. Alignment statistic naming conventions are from the BLAST documentation, see BLAST+ appendices (specifically "outfmt" in table C1). This glossary of common BLAST terms may also be useful in interpreting alignment statistic meaning.

index	field name	data type	description
0	Contig	string	ID/accession for the parent contig/genome sequence.
1	Loc_coordinates	string	Start and end position of the candidate locus (relative to the parent sequence).
2	Name	string	Feature name/label. This is will be identical to "Description" (index 8) if `parse_descriptions` is `True`.
3	Coordinates	string	Start and end position of this feature, relative to the parent sequence.
4	ORFID	string	A unique ID given to this feature, primarily for internal use. Only applies to features that are genes.
5	Strand	signed int	Specifies if the feature was found in the forward (1) or backward (-1) direction. Only applied to features that are genes.
6	Accession	string	ID/accession for the reference sequence that had the best alignment (by e-value) with this feature's translated sequence.
7	E_val	float	The e-value score for the best alignment for this feature.
8	Description	string	A description of this putative feature, parsed from the defline of best aligned reference sequence.
9	Sequence	string	The (translated) amino acid sequence for this feature.
10	Bitscore	float	The bitscore for the best alignment for this feature.
11	Rawscore	int	The raw score for the best alignment for this feature.
12	Aln_len	int	The length of the best scoring alignment, in base pairs.
13	Pident	float	The fraction of identical positions in the best alignment.
14	Nident	int	The number of identical positions in the best alignment.
15	Mismatch	int	The number of mismatched positions in the best alignment.
16	Positive	int	The number of positive-scoring matches in the best alignment.
17	Gapopen	int	The number of gap openings.
18	Gaps	int	Total number of gaps in the alignment.
19	Ppos	float	Percentage of positive scoring matches.
20	Qcovhsp	int	Query coverage per HSP. That is, the fraction of the query (this feature's translated amino acid sequence) that was covered in the best alignment.
21	Contig_filename	string	The input data (genomic sequence(s)) file path.

1.4 API Reference

1.4.1 Gene Finder

gene_finder.pipeline

class gene_finder.pipeline.**Pipeline**

Coordinates protein (or nucleic acid) searches to find gene clusters of interest in genomic/metagenomic data.

add_seed_step(*db*, *name*, *e_val*, *blast_type*, *sensitivity=None*, *parse_descriptions=True*, *blast_path=None*, ***kwargs*)

Find genomic regions that contain at least one "seed" sequence.

Parameters

- **db** (*str*) – Path to the target (seed) protein database.
- **name** (*str*) – A unique name/ID for this step in the pipeline.
- **e_val** (*float*) – Expect value to use. Only keep hits with a an equivalent or better (lower) score.
- **blast_type** (*str*) – Specifies which search program to use. This can be either "PROT" (blastp), "PSI" (psiblast), "mmseqs" (mmseqs2), or "diamond" (diamond).
- **sensitivity** (*str*) – Sets the sensitivity param for mmseqs and diamond (does nothing if BLAST is the seach type).
- **parse_descriptions** (*bool, optional*) – By default, reference protein descriptions (from fasta headers) are parsed for gene name labels; specifically, descriptions are split on whitespace characters and the second item is used for the label. Make this false to simply use the whole protein description for the label (i.e everything after the first whitespace in the header). If using this option with NCBI BLAST, DO NOT use the `-parse_seqids` flag when creating protein databases with **makeblastdb**.
- **blast_path** (*string, optional*) – Path to the blastp/mmseqs/diamond program, if not using the system default.
- ****kwargs** – These can be any additional BLAST parameters, specified as key-value pairs. Note that certain parameters are not allowed, mainly those that control output formatting. Currently only supported for blastp/psiblast; if blast_type is set to mmseqs or diamond, kwargs will be silently ignored.

Note: This should be the first step added to a *gene_finder.pipeline.Pipeline* object. Additional gene finding steps can be added in any order.

---

add_seed_with_coordinates_step(*db*, *name*, *e_val*, *blast_type*, *sensitivity=None*, *parse_descriptions=True*, *start=None*, *end=None*, *contig_id=None*, *blast_path=None*, ***kwargs*)

Define a genomic region of interest with coordinates instead of a seed sequence.

An alternative to *gene_finder.pipeline.Pipeline.add_seed_step()*. Most useful for re-annotating putative systems of interest, where the region coordinates are already known.

Parameters

- **db** (*str*) – Path to the target database to search against.
- **name** (*str*) – A unique name/ID for this step in the pipeline.

- **e_val** (*float*) – Expect value to use. Only keep hits with an equivalent or better (lower) score.
- **blast_type** (*str*) – Specifies which search program to use. This can be either "PROT" (blastp), "PSI" (psiblast), "mmseqs" (mmseqs2), or "diamond" (diamond).
- **sensitivity** (*str*) – Sets the sensitivity param for mmseqs and diamond (does nothing if BLAST is the seach type).
- **parse_descriptions** (*bool, optional*) – By default, reference protein descriptions (from fasta headers) are parsed for gene name labels; specifically, descriptions are split on whitespace characters and the second item is used for the label. Make this false to simply use the whole protein description for the label (i.e everything after the first whitespace in the header). If using this option with NCBI BLAST, DO NOT use the `-parse_seqids` flag when creating protein databases with **makeblastdb**.
- **start** (*int*) – Defines the beginning of the region to search, in base pairs (bp). If no start position is given the first (zero indexed) position in the genome/contig is used.
- **end** (*int*) – Defines the end of the region to search, in base pairs (bp). If no end position is given the last position in the contig is used.
- **contig_id** (*string, optional*) – An identifier for the contig to search. If no ID is given, the pipeline will search every contig in the input file using the coordinates specified. Note that the contig ID is defined as the substring between the ">" character and the first " " character in the contig header.
- **blast_path** (*string, optional*) – Path to the blastp/mmseqs/diamond program, if not using the system default.
- ****kwargs** – These can be any additional BLAST parameters, specified as key-value pairs. Note that certain parameters are not allowed, mainly those that control output formatting. Currently only supported for blastp/psiblast; if blast_type is set to mmseqs or diamond, kwargs will be silently ignored.

add_filter_step(*db*, *name*, *e_val*, *blast_type*, *min_prot_count=1*, *sensitivity=None*, *parse_descriptions=True*, *blast_path=None*, ***kwargs*)

Add a step to search candidate regions for target sequences, and filter out candidates that do not have at least `min_prot_count` matching sequences.

Parameters

- **db** (*str*) – Path to the target protein sequence database.
- **name** (*str*) – A unique name/ID for this step in the pipeline.
- **e_val** (*float*) – Expect value to use. Only keep hits with a an equivalent or better (lower) score.
- **blast_type** (*str*) – Specifies which search program to use. This can be either "PROT" (blastp), "PSI" (psiblast), "mmseqs" (mmseqs2), or "diamond" (diamond).
- **min_prot_count** (*int, optional*) – Minimum number of hits needed to keep each candidate.
- **sensitivity** (*str*) – Sets the sensitivity param for mmseqs and diamond (does nothing if BLAST is the seach type).
- **parse_descriptions** (*bool, optional*) – By default, reference protein descriptions (from fasta headers) are parsed for gene name labels; specifically, descriptions are split on whitespace characters and the second item is used for the label. Make this false to simply use the whole protein description for the label (i.e everything after the first whitespace in

the header). If using this option with NCBI blast, DO NOT use the `-parse_seqids` flag when creating protein databases with **makeblastdb**.

- **blast_path** (*string, optional*) – Path to the blastp/mmseqs/diamond program, if not using the system default.
- ****kwargs** – These can be any additional BLAST parameters, specified as key-value pairs. Note that certain parameters are not allowed, mainly those that control output formatting. Currently only supported for blastp/psiblast; if blast_type is set to mmseqs or diamond, kwargs will be silently ignored.

add_blast_step(*db*, *name*, *e_val*, *blast_type*, *sensitivity=None*, *parse_descriptions=True*, *blast_path=None*, ***kwargs*)

Add a non-filtering search step to the pipeline. That is, search each candidate for target sequences without applying any filtering logic. This is most useful for annotating candidates for non-essential or ancillary genes.

Parameters

- **db** (*str*) – Path to the target protein sequence database.
- **name** (*str*) – A unique name/ID for this step in the pipeline.
- **e_val** (*float*) – Expect value to use. Only keep hits with a an equivalent or better (lower) score.
- **blast_type** (*str*) – Specifies which search program to use. This can be either "PROT" (blastp), "PSI" (psiblast), "mmseqs" (mmseqs2), or "diamond" (diamond).
- **sensitivity** (*str*) – Sets the sensitivity param for mmseqs and diamond (does nothing if BLAST is the seach type).
- **parse_descriptions** (*bool, optional*) – By default, reference protein descriptions (from fasta headers) are parsed for gene name labels; specifically, descriptions are split on whitespace characters and the second item is used for the label. Make this false to simply use the whole protein description for the label (i.e everything after the first whitespace in the header). If using this option with NCBI BLAST, DO NOT use the `-parse_seqids` flag when creating protein databases with **makeblastdb**.
- **blast_path** (*string, optional*) – Path to the blastp/mmseqs/diamond program, if not using the system default.
- ****kwargs** – These can be any additional BLAST parameters, specified as key-value pairs. Note that certain parameters are not allowed, mainly those that control output formatting. Currently only supported for blastp/psiblast; if blast_type is set to mmseqs or diamond, kwargs will be silently ignored.

add_crispr_step()

Add a step to search for CRISPR arrays using PILER-CR.

add_blastn_step(*db*, *name*, *e_val*, *parse_descriptions=False*, *blastn_path='blastn'*, ***kwargs*)

Add a step to do nucleotide BLAST.

Parameters

- **db** (*str*) – Path to the target protein sequence database.
- **name** (*str*) – A unique name/ID for this step in the pipeline.
- **e_val** (*float*) – Expect value to use. Only keep hits with a an equivalent or better (lower) score.

- **parse_descriptions** (*bool, optional*) – By default, reference protein descriptions (from fasta headers) are parsed for gene name labels; specifically, descriptions are split on whitespace characters and the second item is used for the label. Make this false to simply use the whole protein description for the label (i.e everything after the first whitespace in the header). If using this option with NCBI BLAST, DO NOT use the `-parse_seqids` flag when creating protein databases with **makeblastdb**.
- **blast_path** (*string, optional*) – Path to the blastn program, if not using the system default.
- ****kwargs** – These can be any additional BLAST parameters, specified as key-value pairs. Note that certain parameters are not allowed, mainly those that control output formatting. Currently only supported for blastp/psiblast; if blast_type is set to mmseqs or diamond, kwargs will be silently ignored.

run(*data*, *job_id=None*, *output_directory=None*, *min_prot_len=60*, *span=10000*, *record_all_hits=False*, *incremental_output=False*, *starting_contig=None*, *gzip=False*) → dict

Execute each step in the pipeline, in the order they were added.

Parameters

- **data** (*str*) – Path to the input data file. Can be a single- or multi-sequence file in fasta format.
- **job_id** (*str, optional*) – A unique ID to prefix all output files. If no ID is given, the string "gene_finder" will be used as the prefix. In any case, results from the pipeline are written to the file <prefix>_results.csv.
- **output_directory** (*str, optional*) – The directory to write output data files to. If no directory is given then the current (working) directory is used.
- **min_prot_len** (*int, optional*) – Minimum ORF length (aa). Default is 60.
- **span** (*int, optional*) – Length (nt) upsteam and downstream of each seed hit to keep. Defines the aproximate size of the genomic neighborhoods that will be used as the search space after the seed step.
- **record_all_hits** (*bool, optional*) – Write data about all genes found (even discarded ones) to the file <job_id>_hits.json, grouped by contig. Note that this contains much of the same information as is in the results CSV file; nevertheless, it may be useful for analysis or troubleshooting a search.
- **incremental_output** (*bool, optional*) – Write results to disk after each contig is processed. Using this option also creates a checkpoint file that gives the ID of the contig that is currently being processed; if the job finishes successfully, this file will be automatically cleaned up. This feature is especially useful for long-running jobs.
- **starting_contig** (*bool, optional*) – The sequence identifier of the contig where the run should begin. In other words, skip over records in the input file until the specified contig is reached, and then run the pipeline as normal. This is usually used in conjunction with `incremental_output`.
- **gzip** (*bool, optional*) – Was this file compressed with gzip?

Returns Candidate systems, grouped by contig id and genomic location.

Return type dict

1.4.2 Operon Analyzer

The following modules comprise the core Operon Analyzer functionality.

operon_analyzer.genes

class operon_analyzer.genes.**Feature**(*name: str*, *coordinates: Tuple[int, int]*, *orfid: str*, *strand: Optional[int]*, *accession: str*, *e_val: Optional[float]*, *description: str*, *sequence: str*, *bit_score: Optional[int] = None*, *raw_score: Optional[int] = None*, *aln_len: Optional[int] = None*, *pident: Optional[float] = None*, *nident: Optional[float] = None*, *mismatch: Optional[float] = None*, *positive: Optional[float] = None*, *gapopen: Optional[int] = None*, *gaps: Optional[int] = None*, *ppos: Optional[float] = None*, *qcovhsp: Optional[int] = None*)

Represents a gene or CRISPR repeat array. This is used internally by *operon_analyzer.genes.Operon*, but appears in the auto-generated documentation for reference.

class operon_analyzer.genes.**Operon**(*contig: str*, *contig_filename: str*, *start: int*, *end: int*, *features: List[*operon_analyzer.genes.Feature*]*)

Provides access to features that were found in the same genomic region, which presumably comprise an actual operon. Whether this is true in reality must be determined by the user, if that is meaningful to them.

set_sequence(*sequence: Bio.Seq.Seq*)

Stores the nucleotide sequence of the operon.

property feature_region_sequence: str

Returns the nucleotide sequence of the operon, excluding the regions outside of the outermost Features.

property all_genes

Iterates over all genes (i.e. not CRISPR arrays) in the operon regardless of whether it's been ignored.

property all_features

Iterates over all features in the operon regardless of whether it's been ignored.

property feature_names

Iterates over the name of each feature in the operon

get(*feature_name: str*, *regex=False*) → List[*operon_analyzer.genes.Feature*]

Returns a list of every Feature with a given name.

get_unique(*feature_name: str*, *regex=False*) → Optional[*operon_analyzer.genes.Feature*]

Returns a Feature or None if there is more than one Feature with the same name

as_str() → str

Writes an Operon back out in the same CSV format that gene_finder produces. The text won't be completely identical in the case where floats have trailing decimals, or zero values in scientific format are recast as a simple float.

operon_analyzer.rules

class operon_analyzer.rules.**SerializableFunction**(*name: str*, *function: Callable*, **args*, *custom_repr: Optional[str] = None*)

A base class for functions that we need to be able to serialize. Do not instantiate this directly.

class operon_analyzer.rules.**Rule**(*name: str*, *function: Callable*, **args*, *custom_repr: Optional[str] = None*)

Defines a requirement that elements of an operon must adhere to.

evaluate(*operon:* operon_analyzer.genes.Operon) → bool

Determine if an operon adheres to this rule.

class operon_analyzer.rules.**Result**(*operon:* operon_analyzer.genes.Operon)

Records which rules an operon passed and handles serialization of the data. Also makes it easy to run follow-up queries.

add_passing(*rule:* operon_analyzer.rules.Rule)

Mark this rule as being one that the operon passed.

add_failing(*rule:* operon_analyzer.rules.Rule)

Mark this rule as being one that the operon failed.

property is_passing: bool

Declares whether the given Operon adhered to all given Rules.

class operon_analyzer.rules.**Filter**(*name: str*, *function: Callable*, **args*, *custom_repr: Optional[str] = None*)

A function that will be run on an Operon that marks Features as being ignorable for the purposes of evaluating RuleSets.

run(*operon:* operon_analyzer.genes.Operon)

Mark Features in the Operon as ignored if they don't pass the filter. This will prevent them from being taken into account during Rule evaluation and (by default) during visualization.

class operon_analyzer.rules.**FilterSet**

Stores functions that take an Operon and mark individual Features as ignored in case we think they are not actually worth taking into account when evaluating rules. Features can be ignored for multiple reasons.

must_be_within_n_bp_of_anything(*distance_bp: int*)

If a feature is very far away from anything it's probably not part of an operon.

must_be_within_n_bp_of_feature(*feature_name: str*, *distance_bp: int*, *regex: bool = False*)

There may be situations where two features always appear near each other in functional operons.

pick_overlapping_features_by_bit_score(*minimum_overlap_threshold: float*)

If two features overlap by more than `minimum_overlap_threshold`, the one with the lower bit score is ignored.

custom(*filt:* operon_analyzer.rules.Filter)

Add a rule with a user-defined function.

evaluate(*operon:* operon_analyzer.genes.Operon)

Run the filters on the operon and set Features that fail to meet the requirements to be ignored.

Parameters **operon** – The *operon_analyzer.genes.Operon* object whose features will be evaluated.

class operon_analyzer.rules.**RuleSet**

Creates, stores and evaluates *operon_analyzer.rules.Rule* s that an operon must adhere to.

exclude(*feature_name: str*, *regex: bool = False*)

Forbid the presence of a particular feature.

require(*feature_name: str*, *regex: bool = False*)
Require the presence of a particular feature.

max_distance(*feature1_name: str*, *feature2_name: str*, *distance_bp: int*, *closest_pair_only: bool = False*, *regex: bool = False*)
The two given features must be no further than `distance_bp` base pairs apart. If there is more than one match, all possible pairs must meet the criteria, unless `closest_pair_only` is True in which case only the closets pair is considered.

at_least_n_bp_from_anything(*feature_name: str*, *distance_bp: int*, *regex=False*)
Requires that a feature be at least `distance_bp` base pairs away from any other feature. This is mostly useful for eliminating overlapping features.

at_most_n_bp_from_anything(*feature_name: str*, *distance_bp: int*, *regex: bool = False*)
A given feature must be within `distance_bp` base pairs of another feature. Requires exactly one matching feature to be present. Returns False if the given feature is the only feature.

same_orientation(*exceptions: Optional[List[str]] = None*)
All features in the operon must have the same orientation.

contains_any_set_of_features(*sets: List[List[str]]*)
Returns True if the operon contains features with all of the names in at least one of the lists. Useful for determining if an operon contains all of the essential genes for a particular system, for example.

contains_exactly_one_of(*feature1_name: str*, *feature2_name: str*, *regex: bool = False*)
An exclusive-or of the presence of two features. That is, one of the features must be present and the other must not.

contains_at_least_n_features(*feature_names: List[str]*, *feature_count: int*, *count_multiple_copies: bool = False*)
The operon must contain at least `feature_count` features in the list. By default, a matching feature that appears multiple times in the operon will only be counted once; to count multiple copies of the same feature, set `count_multiple_copies` to True.

contains_group(*feature_names: List[str]*, *max_gap_distance_bp: int*, *require_same_orientation: bool*)
The operon must contain a contiguous set of features (in any order) separated by no more than `max_gap_distance_bp`. Optionally, the user may require that the features must all have the same orientation.

maximum_size(*feature_name: str*, *max_bp: int*, *all_matching_features_must_pass: bool = False*, *regex: bool = False*)
The operon must contain at least one feature with `feature_name` with a size (in base pairs) of `max_bp` or smaller. If `all_matching_features_must_pass` is True, every matching Feature must be at least `max_bp` long.

minimum_size(*feature_name: str*, *min_bp: int*, *all_matching_features_must_pass: bool = False*, *regex: bool = False*)
The operon must contain at least one feature with `feature_name` with a size (in base pairs) of `min_bp` or larger. If `all_matching_features_must_pass` is True, every matching Feature must be at least `min_bp` long.

custom(*rule:* operon_analyzer.rules.Rule)
Add a rule with a user-defined function.

evaluate(*operon:* operon_analyzer.genes.Operon) → *operon_analyzer.rules.Result*
See if an operon adheres to all rules.

Parameters **operon** – The *operon_analyzer.genes.Operon* object to evaluate.

operon_analyzer.analyze

operon_analyzer.analyze.**analyze**(*input_lines: IO[str]*, *ruleset:* operon_analyzer.rules.RuleSet, *filterset: Optional[*operon_analyzer.rules.FilterSet*] = None*, *output: Optional[IO] = None*)

Takes a handle to the CSV generated by *gene_finder.pipeline.Pipeline* and a *operon_analyzer.rules.RuleSet* object, and produces text that describes which operons adhered to those rules. If an operon fails any of the rules, the exact rules will be enumerated.

operon_analyzer.analyze.**evaluate_rules_and_reserialize**(*input_lines: IO[str]*, *ruleset:* operon_analyzer.rules.RuleSet, *filterset: Optional[*operon_analyzer.rules.FilterSet*] = None*, *output: Optional[IO] = None*)

Takes a handle to the CSV generated by *gene_finder.pipeline.Pipeline* and a *operon_analyzer.rules.RuleSet* object, and writes passing operons back to stdout.

operon_analyzer.analyze.**load_analyzed_operons**(*f: IO[str]*) → Iterator[Tuple[str, int, int, str]]

Loads and parses the data from the output of *operon_analyzer.analyze.analyze()*. This is typically used for analyzing or visualizing candidate operons.

operon_analyzer.analyze.**group_similar_operons**(*operons: List[*operon_analyzer.genes.Operon*]*, *load_sequences: bool = True*)

Groups operons together if the nucleotide sequences bounded by their outermost features (represented by *operon_analyzer.genes.Feature* objects) are identical. If `load_sequences` is True, the nucleotide sequence of each operon will be loaded from disk as it is encountered.

Returns A representative *operon_analyzer.genes.Operon* object for each group.

Return type list

operon_analyzer.analyze.**deduplicate_operons_approximate**(*operons: Iterator[*operon_analyzer.genes.Operon*]*) → List[*operon_analyzer.genes.Operon*]

Deduplicates Operons by examining the names and sequences of their features (represented by *operon_analyzer.genes.Feature* objects) and the sizes of the gaps between them. This is an approximate algorithm: false positives are possible when the nucleotide sequence varies between the Features (without changing the total number of base pairs) or if there are silent mutations in the Feature CDS. However, it is much faster than the exact method.

Returns A representative *operon_analyzer.genes.Operon* object for each group.

Return type list

operon_analyzer.analyze.**dedup_supersets**(*operons: List[*operon_analyzer.genes.Operon*]*) → List[*operon_analyzer.genes.Operon*]

If the same inputs are searched with *gene_finder.pipeline.Pipeline* using an expanded database, the new results will be either exactly identical to the previous results, or will be supersets of the old results.

This function takes all operons, and removes ones with identical accession IDs and contig coordinates, where the smaller operon's features are all contained in the larger one.

Returns The non-redundant *operon_analyzer.genes.Operon* objects.

Return type list

operon_analyzer.analyze.**cluster_operons_by_feature_order**(*operons: Iterator[*operon_analyzer.genes.Operon*]*)

Organizes all operons into a dictionary based on the order/identity of their features (represented by *operon_analyzer.genes.Feature* objects). Cases where the overall order is inverted are considered to be

the same. The keys of the dictionary are the dash-delimited feature names, with one of the two orientations (if both exist) arbitrarily chosen. If there are ignored features, they will not appear in the key.

Returns The resulting *operon_analyzer.genes.Operon* clusters.

Return type dict

operon_analyzer.visualize

operon_analyzer.visualize.**plot_operons**(*operons: List[*operon_analyzer.genes.Operon*]*, *output_directory: str*, *plot_ignored: bool = True*, *color_by_blast_statistic: Optional[str] = None*, *feature_colors: Optional[dict] = {}*, *nucl_per_line: Optional[int] = None*, *show_accession: bool = False*, *show_description: bool = False*)

Takes *operon_analyzer.genes.Operon* objects and saves plots of them to disk.

Parameters

- **operons** (*list*) – Operons to be plotted.
- **output_directory** (*str*) – Path to the directory to save operon plots to.
- **plot_ignored** (*bool, optional*) – Toggles plotting of features that were marked as ignorable by *operon_analyzer.rules.FilterSet* .
- **color_by_blast_statistic** (*str, optional*) – Map an alignment quality statistic using the virdis color scale. For a list of alignment statistics captured by Opfi, see *Opfi output format* .
- **feature_colors** (*dict, optional*) – If a labeled database was used during candidate identification, features can be colored accordingly using "label": "feature-color" pairs. For more information about labeling sequence databases, see *Annotating sequence databases* .
- **nucl_per_line** (*int, optional*) – Length (in base pairs) to wrap gene diagrams on.
- **show_accession** (*bool, optional*) – Show the accession number of the best hit for each plotted feature.
- **show_description** (*bool, optional*) – Show the description of the best hit for each plotted feature.

operon_analyzer.visualize.**plot_operon_pairs**(*operons: List[*operon_analyzer.genes.Operon*]*, *other_operons: List[*operon_analyzer.genes.Operon*]*, *output_directory: str*, *color_by_blast_statistic: Optional[str] = None*, *plot_ignored: bool = False*, *feature_colors: Optional[dict] = {}*)

Takes two lists of presumably related Operons, pairs them up such that the pairs overlap the same genomic region, and plots one on top of the other. This allows side-by-side comparison of two different pipeline runs, so that you can, for example, run your regular pipeline, then re-BLAST with a more general protein database like nr, and easily see how the annotations differ.

Parameters

- **operons** (*list*) – Operons to be plotted.
- **other_operons** (*list*) – Related operons to be plotted for comparison.
- **output_directory** (*str*) – Path to the directory to save operon plots to.
- **plot_ignored** (*bool, optional*) – Toggles plotting of features that were marked as ignorable by *operon_analyzer.rules.FilterSet* .

- **color_by_blast_statistic** (*str, optional*) – Map an alignment quality statistic using the virdis color scale. For a list of alignment statistics captured by Opfi, see *Opfi output format* .
- **feature_colors** (*dict, optional*) – If a labeled database was used during candidate identification, features can be colored accordingly using "label": "feature-color" pairs. For more information about labeling sequence databases, see *Annotating sequence databases* .

operon_analyzer.visualize.**make_clustered_stacked_operon_plots**(*operons: Iterable[*operon_analyzer.genes.Operon*]*, *other_operons: Iterable[*operon_analyzer.genes.Operon*]*, *image_directory: str*, *min_count: int = 10*, *plot_ignored: bool = False*, *color_by_blast_statistic: Optional[str] = None*, *feature_colors: Optional[dict] = None*)

Clusters operons and plots them on top of a reannotated version of the same operon. This allows the user to BLAST some set of data with a curated database, then re-BLAST it against a more general database, and compare the two directly in a cluster-specific manner.

If a *operon_analyzer.rules.FilterSet* was used during analysis, that same set should be evaluated on each operon before passing it into this function.

Parameters

- **operons** (*iterable*) – The operons of interest.
- **other_operons** (*iterable*) – Reannotated operons.
- **image_directory** (*str*) – The directory where all subdirectories will be created. Will be created if it does not exist.
- **min_count** (*int, optional*) – Groups must have at least this many systems in order to be plotted. Default is 10.
- **plot_ignored** (*bool, optional*) – Toggles plotting of features that were marked as ignorable by *operon_analyzer.rules.FilterSet* .
- **color_by_blast_statistic** (*str, optional*) – Map an alignment quality statistic using the virdis color scale. For a list of alignment statistics captured by Opfi, see *Opfi output format* .
- **feature_colors** (*dict, optional*) – If a labeled database was used during candidate identification, features can be colored accordingly using "label": "feature-color" pairs. For more information about labeling sequence databases, see *Annotating sequence databases* .
- **nucl_per_line** (*int, optional*) – Length (in base pairs) to wrap gene diagrams on.

operon_analyzer.visualize.**make_clustered_operon_plots**(*analysis_csv: str*, *operons: Iterable[*operon_analyzer.genes.Operon*]*, *image_directory: str*, *min_count: int = 10*, *diff_against_csv: Optional[str] = None*, *plot_ignored: bool = False*, *color_by_blast_statistic: Optional[str] = None*, *feature_colors: Optional[dict] = None*, *nucl_per_line: Optional[int] = None*)

Clusters operons by the order of their features and plots them in separate directories, adding the number of systems to the directory name. Only systems that passed the rules specified as a *operon_analyzer.rules.RuleSet*

object will be eligible to be plotted.

If a *operon_analyzer.rules.FilterSet* was used during analysis, that same FilterSet should be evaluated on each operon before passing it into this function.

Parameters

- **analysis_csv** (*str*) – Path to the CSV file created by *operon_analyzer.analyze.analyze()* .
- **operons** (*iterable*) – The operons of interest.
- **image_directory** (*str*) – The directory where all subdirectories will be created. Will be created if it does not exist.
- **min_count** (*int, optional*) – Groups must have at least this many systems in order to be plotted. Default is 10.
- **diff_against_csv** (*str*) – Path to a CSV file created by operon analyzer. Any clusters in this file will be skipped when clustering operons from analysis_csv. The point of this is to see only new systems when making slight alterations to rules.
- **plot_ignored** (*bool, optional*) – Toggles plotting of features that were marked as ignorable by *operon_analyzer.rules.FilterSet* .
- **color_by_blast_statistic** (*str, optional*) – Map an alignment quality statistic using the virdis color scale. For a list of alignment statistics captured by Opfi, see *Opfi output format* .
- **feature_colors** (*dict, optional*) – If a labeled database was used during candidate identification, features can be colored accordingly using "label": "feature-color" pairs. For more information about labeling sequence databases, see *Annotating sequence databases* .

operon_analyzer.overview

operon_analyzer.overview.**load_counts**(*lines: IO[str]*) → Tuple[Dict[str, int], Dict[str, int], Dict[int, int]]

Takes a stream of the output from *operon_analyzer.analyze.analyze()* and computes some useful statistics. Namely, the number of times each rule was the only one broken for a contig, the number of times each rule was broken regardless of context, and the occurrences of the count of broken rules per contig.

operon_analyzer.reannotation

operon_analyzer.reannotation.**summarize**(*operons: List[*operon_analyzer.genes.Operon*]*, *reannotated_operons: List[*operon_analyzer.genes.Operon*]*)

For each protein in each operon, prints out how frequently it was reannotated to a particular protein. For example, a putative Cas9 may be identified as a transposase when BLASTed with a more expansive database.

operon_analyzer.load

operon_analyzer.load.**load_sequence**(*operon:* operon_analyzer.genes.Operon) → Optional[Bio.Seq.Seq]
Loads the DNA sequence for a given operon's contig.

operon_analyzer.load.**load_gzipped_operons**(*filename: str*) → Iterator[*operon_analyzer.genes.Operon*]
Create a *operon_analyzer.genes.Operon* representation for each candidate present in gzipped output from *gene_finder.pipeline.Pipeline*.

Parameters **filename** (*str*) – Path to the compressed pipeline CSV file.

operon_analyzer.load.**load_operons**(*handle: IO[str]*) → Iterator[*operon_analyzer.genes.Operon*]
Create a *operon_analyzer.genes.Operon* representation for each candidate present in output from *gene_finder.pipeline.Pipeline*.

Parameters **handle** (*IO*) – Handle for the pipeline CSV file.

operon_analyzer.parse

operon_analyzer.parse.**assemble_operons**(*lines: Iterator[Tuple[str, str, str, str, str, str, str, str, str, str]]*) → Iterator[*operon_analyzer.genes.Operon*]
Takes the output from *gene_finder.pipeline.Pipeline* and loads all features, then assembles them into *operon_analyzer.genes.Operon* objects.

To keep things memory efficient while not allowing redundant operons from being loaded, we keep track of the hash of each operon, which is just an integer. Even for very large metagenomic databases this should use less than a gigabyte of memory.

This is a helper function used by the loaders in *operon_analyzer.load*, but appears in the auto-generated documentation for reference.

The next set of modules expose simple functions for dealing with CRISPR array sequences. Presumably, these would only be useful to researchers interested in CRISPR-Cas genomic systems.

operon_analyzer.piler_parse

Parses piler-cr output to extract the sequences of every spacer.

class operon_analyzer.piler_parse.**RepeatSpacer**(*position*, *repeat_len*, *spacer_len*, *sequence*)

property position
Alias for field number 0

property repeat_len
Alias for field number 1

property sequence
Alias for field number 3

property spacer_len
Alias for field number 2

class operon_analyzer.piler_parse.**BrokenSpacer**(*position*, *repeat_len*, *sequence*)

property position
Alias for field number 0

property repeat_len
Alias for field number 1

property sequence
Alias for field number 2

operon_analyzer.piler_parse.**parse_pilercr_output**(*text: str*, *start: int*, *end: int*)
Takes the text of **pilercr** raw output, and a start and end coordinate for an operon's neighborhood, and extracts all spacers in that region.

operon_analyzer.repeat_finder

class operon_analyzer.repeat_finder.**Repeat**(*upstream_sequence*, *upstream_start*, *downstream_sequence*, *downstream_start*)

property downstream_sequence
Alias for field number 2

property downstream_start
Alias for field number 3

property upstream_sequence
Alias for field number 0

property upstream_start
Alias for field number 1

class operon_analyzer.repeat_finder.**GRFResult**(*start*, *end*, *alignment*)

property alignment
Alias for field number 2

property end
Alias for field number 1

property start
Alias for field number 0

operon_analyzer.repeat_finder.**find_inverted_repeats**(*operon: operon_analyzer.genes.Operon*, *buffer_around_operon: int*, *min_repeat_size: int*)
Searches an operon and the DNA flanking it for inverted repeats using **GenericRepeatFinder**. If found, they will be added to the operon as Feature objects with the name "TIR" and the sequence. The strand will be set to 1 for the upstream sequence and -1 for the downstream sequence.

Parameters

- **operon** (*Operon*) – The *operon_analyzer.genes.Operon* object.
- **buffer_around_operon** (*int*) – The number of base pairs on either side of the operon to search in addition to the operon's internal sequence.
- **min_repeat_size** (*int*) – The minimum number of base pairs that an inverted repeat must have.

operon_analyzer.spacers

class operon_analyzer.spacers.**AlignmentResult**(*match_count*, *spacer_sequence*, *contig_sequence*, *contig_start*, *contig_end*, *spacer_alignment*, *contig_alignment*, *comp_string*, *strand*, *spacer_order*, *array_length*)

property array_length
Alias for field number 10

property comp_string
Alias for field number 7

property contig_alignment
Alias for field number 6

property contig_end
Alias for field number 4

property contig_sequence
Alias for field number 2

property contig_start
Alias for field number 3

property match_count
Alias for field number 0

property spacer_alignment
Alias for field number 5

property spacer_order
Alias for field number 9

property spacer_sequence
Alias for field number 1

property strand
Alias for field number 8

operon_analyzer.spacers.**find_self_targeting_spacers**(*operons: List[*operon_analyzer.genes.Operon*]*, *min_matching_fraction: float*, *num_processes: int = 32*)

For each given Operon, this will determine if CRISPR spacers target a location in the operon's parent contig. Matches with more than `min_matching_fraction` homology will be added to the Operon as a Feature named "CRISPR target".

1.5 Contributing

Thank you for your interest in contributing to Opfi. Contribututions can take many forms, including:

- Reporting a bug
- Discussing the current state of the code
- Submitting a fix
- Proposing new features

1.5.1 Issues

Issues should be used to report problems or bugs with the library, to request new features, or to discuss potential changes before PRs are created.

Good bug reports have:

- A summary of the issue
- Specific steps to reproduce the problem, preferably in the form of code examples
- A description of expected behavior
- A description of actual behavior

1.5.2 Pull Requests

In general, we use Github Flow. This means that all changes happen through Pull Requests:

1. Fork the repository to your own Github account
2. Clone the project to your machine
3. Create a branch locally with a succinct but descriptive name
4. Commit changes to the branch
5. Push changes to your fork
6. Open a PR in our repository and include a description of the changes and a reference to the issue the PR addresses

1.5.3 Coding Style

Contributors should attempt to adhere to the PEP8 style guide when possible, although this is not currently strictly enforced. At a minimum, new contributions should follow a style that is consistent with the preexisting code base.

1.5.4 Testing

We use the pytest framework for creating unit tests. Please write tests for any new code contributed to this project.

1.5.5 License

By contributing, you agree that your contributions will be licensed under its MIT License.

1.5.6 Code of Conduct

We take our open source community seriously and hold ourselves and other contributors to high standards of communication. By participating and contributing to this project, you agree to uphold our Code of Conduct.

1.5.7 Attribution

This document was adapted from the General Contributing Guidelines of the rextendr project, and from the contributing template developed by briandk.

PYTHON MODULE INDEX

g

o

INDEX

F

G

I

L

M

O

P

R

S

U