cblaster

Release 1.3.18

Feb 16, 2023

Contents

Welcome to cblaster's documentation!

cblaster is a tool for identifying co-located hits in BLAST searches against NCBI sequence databases. It leverages the NCBI's public APIs to facilitate fully remote searches, requiring no setup of local search databases.

If you find `cblaster` helpful, please cite:

```
Cameron L.M. Gilchrist, Thomas J. Booth, Bram van Wersch, Liana van Grieken, Marnix H.
↪ Medema, Yit-Heng Chooi (2020).
cblaster: a remote search tool for rapid identification and visualisation of␣
↪homologous gene clusters.
bioRxiv 2020.11.08.370601; doi: https://doi.org/10.1101/2020.11.08.370601
```

To view an example of what cblaster can produce, click here.

CHAPTER 1

Features

- Fully remote searches against public NCBI sequence databases
- One command to generate local search databases from many genomes
- Easy to use graphical user interface (GUI)
- Fully interactive visualisations

CHAPTER 2

User guide

2.1 User Guide

The `cblaster` software was written by Cameron Gilchrist with contributions from Dr. Thomas Booth and Dr. Yit-Heng Chooi.

If you find `cblaster` helpful, please cite:

```
Cameron L.M. Gilchrist, Thomas J. Booth, Bram van Wersch, Liana van Grieken, Marnix H.
↪ Medema, Yit-Heng Chooi (2020).
cblaster: a remote search tool for rapid identification and visualisation of␣
↪homologous gene clusters.
bioRxiv 2020.11.08.370601; doi: https://doi.org/10.1101/2020.11.08.370601
```

If you have any questions, would like to report a bug, or have any suggestion on how we could improve cblaster, please contact:

- cameron.gilchrist@research.uwa.edu.au
- thomas.booth@uwa.edu.au
- yitheng.chooi@uwa.edu.au

2.1.1 Table of Contents

What is `cblaster`

`cblaster` is a tool for finding clusters of co-located sequences using BLAST searches.

By providing `cblaster` with amino acid sequences of interest, it can search a sequence database to find instances of genes encoding related proteins clustered on a specific scaffold. The software was developed as a spiritual successor to software such as MultiGeneBlast and, to aid in the discovery of natural product biosynthetic gene clusters in bacteria and fungi.

It can, of course, be used to search for any group of clustered genes.

`cblaster` offers a number of useful functions:

- `cblaster search` allows users to query a FASTA file containing protein sequences against the NCBI database or a local database to identify instances of collocation. A number of graphical and textual outputs are available for quick analysis or use in downstream applications.
- `cblaster makedb` enables the user to rapidly create local databases from GenBank files for future searches.
- `cblaster gne` can be used to analyse the effect of varying the intergenic distances on the search output. This can be used to evaluate the robustness of outputs and gain insights into the distribution of gene cluster sizes.
- `cblaster extract` allows the user to quickly produce FASTA files containing the protein sequences of homologues of interest for downstream analysis.

Detailed information on how to install `cblaster` and use the above modules can be found in this guide.

Installation and Quick Start

Installing Python for Windows

In order to use `cblaster` you will first need to install Python on your computer. Go to python.org/downloads/ and download the latest version of Python (3.8.5 at the time of writing). This will initiate the download of the Python installer (python-x.x.x.exe).

Locate the installer in your downloads folder, run the program and follow the wizard. **Make sure you tick the box 'Add Python x.x to PATH'.** This ensures that `cblaster` is available for you to use directly from the terminal. It is not selected by default and you will have to do this step manually if you do not check the box here.

Once the installer has finished, you can try to run Python in PowerShell to verify that it has been installed correctly. To open PowerShell in Windows, open a folder, Shift+Right click and select the option 'Open PowerShell window here. . . '. With PowerShell open, type `python` and press enter. If python has installed correctly, the interactive shell should be launched, and the version number should be displayed in the console like so:

```
Python 3.8.5 (default, Jul 20 2020, 17:41:41)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The Python package installer tool, `pip` is installed alongside Python. This is necessary to install `cblaster`, so verify that it is installed by typing `pip --version` in PowerShell as above. This should print the version information as well as the Python version and the location like so:

```
pip 19.2.3 from c:\...\pip (python 3.8)
```

Installing DIAMOND

`cblaster` uses `DIAMOND` to perform local searches. This can be freely obtained from http://www.diamondsearch.org/index.php. Make sure to download and install DIAMOND prior to installing cblaster.

Installing, uninstalling and updating cblaster

To install `cblaster`, simply input the command:

```
pip install cblaster
```

This will install `cblaster` as well as all of its dependencies.

Should you decide to uninstall cblaster, this can also be done using pip:

```
pip uninstall cblaster
```

Note: If a new version of cblaster is available, you can simply uninstall and reinstall the module as above to access the newer version.

Running your first search

Once `cblaster` has been installed, running a search is as simple as providing a collection of query amino acid sequences in a FASTA file, like so:

```
cblaster search -qf query.fasta
```

Visualisations can be generated using the `-p` or `--plot` argument:

```
cblaster search -qf query.fasta -p plot.html
```

Note: if no file name is provided, the plot will be dynamically served using Python's built in HTTP server. The plot will be exactly the same, but it will not generate a static HTML file that can be shared around.

Search sessions can be saved for later re-use using the `-s` or `--session` argument:

```
cblaster search -qf query.fasta -p plot.html -s session.json
```

Note: a session is saved as a JavaScript Object Notation (JSON) format file. This is essentially just a dump of all the code objects, as well as search parameters, used during a `cblaster` search. If you provide a pre-existing session file, `cblaster` will attempt to load it **instead** of performing a new search.

That is all you need to know about the basic usage of `cblaster`. However, there are many more ways to tweak and run the program to suit your needs which are further explored in the following sections.

Pre-search configuration using the `config` module

The NCBI requires that you provide some identification before using their services in order to prevent abuse. This can be an e-mail address, or more recently, an API key (https://ncbiinsights.ncbi.nlm.nih.gov/2017/11/02/new-api-keys-for-the-e-utilities/).

You can use the `config` module to set these parameters for `cblaster` searches (you'll only have to do this once!). This module will save a file, `config.ini`, wherever your operating system stores configuration files (for example, in Linux it will be saved in ~/.local/config/cblaster). When you run remote searches in `cblaster`, it will first check to see if it can find this file, and then if an e-mail address or API key is saved; if they are not found, `cblaster` will throw an error.

To set an e-mail address:

```
$ cblaster config --email "foo@bar.com"
```

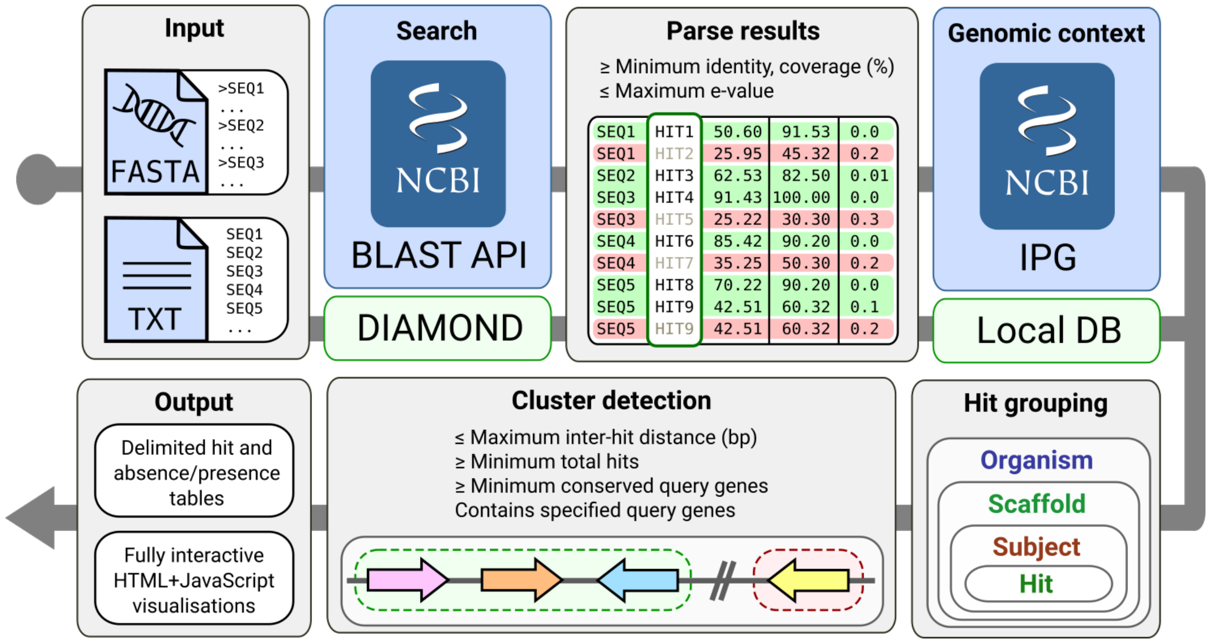
. . . or an API key:

```
$ cblaster config --api_key <your API key>
```

Running a `cblaster` search using the `search` module

The `cblaster` search workflow

Both local and remote `cblaster` searches proceed through a similar workflow, which is depicted in the following figure:


Input
FASTA
>SEQ1
...
>SEQ2
...
>SEQ3
...
TXT
SEQ1
SEQ2
SEQ3
SEQ4
SEQ5
...
Search
NCBI
BLAST API
DIAMOND
Parse results
≥ Minimum identity, coverage (%)
≤ Maximum e-value
SEQ1 HIT1 50.60 91.53 0.0
SEQ1 HIT2 25.95 45.32 0.2
SEQ2 HIT3 62.53 82.50 0.01
SEQ3 HIT4 91.43 100.00 0.0
SEQ3 HIT5 25.22 30.30 0.3
SEQ4 HIT6 85.42 90.20 0.0
SEQ4 HIT7 35.25 50.30 0.2
SEQ5 HIT8 70.22 90.20 0.0
SEQ5 HIT9 42.51 60.32 0.1
SEQ5 HIT9 42.51 60.32 0.2
Genomic context
NCBI
IPG
Local DB
Output
Delimited hit and absence/presence tables
Fully interactive HTML+JavaScript visualisations
Cluster detection
≤ Maximum inter-hit distance (bp)
≥ Minimum total hits
≥ Minimum conserved query genes
Contains specified query genes
Hit grouping
Organism
Scaffold
Subject
Hit

First, query sequences are searched against the NCBI's BLAST API or a local DIAMOND database, in remote (blue background) and local (green background) modes, respectively. BLAST hits are filtered according to user defined quality thresholds.

In remote mode, each hit is then queried against the NCBI's Identical Protein Groups (IPG) resource, which, as its name suggests, groups proteins sharing identical amino acid sequence as an anti-redundancy measure. The resulting IPG table contains source genomic coordinates for each hit protein sequence, which `cblaster` uses to group them by their corresponding organism, scaffold and subject sequences.

In local mode, a special local database is created for this purpose (see *Creating local sequence databases with the makedb module* for more information). Finally, `cblaster` scans the scaffolds on each organism for clustered BLAST hits and generates sumamry output tables and visualisations.

Warning: Before running a `cblaster` search, you must run the `cblaster config` module to provide either an e-mail address **or** an NCBI API key. See *Pre-search configuration using the config module* for how to do this.

In order to run a `cblaster` search, you will need to point the module to a collection of sequences to be used as queries. These can be provided in two ways:

1. A FASTA format file containing amino acid sequences
2. A list of valid NCBI sequence identifiers (e.g. accession, GI number)

If using a FASTA file, it can be passed to `cblaster` using the `-qf/--query_file` argument:

```
$ cblaster search -qf myFile.fasta
```

Conversely, sequence identifiers are passed using the `-qi/--query_ids` argument. These can either be given either in a newline-separated text file:

```
$ cat myFile.fasta
query1
query2
...
$ cblaster search -qi myFile.txt
```

Or directly to the command line:

```
$ cblaster search -qi query1 query2 ...
```

Coincidentally, both of the above commands are fully valid search commands, and will launch a remote search against the NCBI using the specified sequences. See *Remote searches against NCBI sequence databases* for more details on remote searches.

Searches against local sequence data

To run a local search, you will need to specify as such using the `-m/--mode` argument, as well as provide both a DIAMOND search database and a `cblaster` SQL database (see *Creating local sequence databases with the makedb module* for details on how to create these files). However, only the DIAMOND database has to be specified in the command: `cblaster` will automatically look for a SQL database with the same name and `.sqlite3` suffix. An example command might look like this:

```
$ cblaster search -m local -db myDB.dmnd -qf myFile.gbk
```

Functional domain searches using HMMER

To run a domain search, you need to specify the search mode as `hmm`, provide an array of query Pfam domain profile names, a FASTA file containing sequences to be searched (produced using the `makedb` module, see *Creating local sequence databases with the makedb module* for details) and the path to a folder containing a copy of the Pfam database.

For example:

```
$ cblaster search -m hmm -qp PF00001 PF00002 -db myDb.fasta -pfam pfamFolder/
```

This will extract the specified domain profiles (`PF00001` and `PF00002`) from the Pfam database and search the sequences in `myDb.fasta` for any domain hits.

Note that like in local searches, `cblaster` expects an SQL database in the same location as the FASTA file, with the same name and `.sqlite3` suffix. Additionally, `cblaster` requires two Pfam database files:

Pfam-A.hmm.gz	Main database file containing HMM profiles
Pfam-A.hmm.dat.gz	File used for looking up domain families from query accessions

The latest versions of these files are automatically downloaded when `cblaster` is given the path to a folder which does not contain them.

Remote searches against NCBI sequence databases

Remote search is the default mode in `cblaster`. As such, in the basic search example:

```
$ cblaster search -qf query.fasta
```

The sequences in `query.fasta` are loaded in and searched remotely. `cblaster` provides several useful options specifically for remote searches.

By default, remote searches will be performed against the NCBI's `nr` database. Alternative databases can be specified using the `-d/--database` argument, for example:

```
$ cblaster search -qf query.fasta -db refseq_protein
```

`cblaster` currently only supports protein sequence searches using `BLASTp`, so you should choose protein sequence databases (e.g. nr, refseq_protein, swissprot, pdbaa).

If `cblaster` has been interrupted somehow during a remote search (i.e. search started but program is stopped before a session can be saved), it can be resumed using the Request Identifier (RID). Every remote search is automatically assigned an RID which can be used to retrieve results up to 36 hours after they have completed. This is reported to the screen when a `cblaster` search starts:

```
$ cblaster search -qf query.fasta
[13:43:16] INFO - Starting cblaster in remote mode
[13:43:16] INFO - Launching new search
[13:43:20] INFO - Request Identifier (RID): RAV3P2F3014
[13:43:20] INFO - Request Time Of Execution (RTOE): 13s
[13:43:33] INFO - Checking search status...
...
```

`cblaster` can resume a search from this RID using the `--rid` argument:

```
$ cblaster search -qf query.fasta --rid RAV3P2F3014
[13:56:21] INFO - Starting cblaster in remote mode
[13:56:21] INFO - Polling NCBI for completion status
[13:56:21] INFO - Checking search status...
[13:56:23] INFO - Search has completed successfully!
[13:56:23] INFO - Retrieving results for search RAV3P2F3014
...
```

Warning: The NCBI prioritises searches started through it's interactive web interface over searches launched via the BLAST API in `cblaster`. This means that, particularly for searches that return a lot of results, searches can take a very long time to complete (hours!). In this case, start a search using the BLAST website (https://blast.ncbi.nlm.nih.gov), make a note of the RID, and pass that to `cblaster` using the `--rid` argument, as well as the file containing your query sequences using the `-qf/--query_file` argument.

Finally, NCBI allows for pre-filtering of search databases using NCBI Entrez search queries. Entrez is the NCBI's text search and retrieval system for all of the databases they provide. The most obvious way to use this in `cblaster` is to filter based on specific taxonomic areas of interest to narrow down the result set. This also has the added benefit of significantly reducing search run times. For example, we can filter the `nr` database for only fungal sequences by providing an organism Entrez search term using the fungi NCBI taxonomy ID (4751) with the `-eq/--entrez_query` argument:

```
$ cblaster search -qf query.fasta -eq "txid4751[orgn]"
```

Note: It is best to ensure your search term is enclosed in speech marks such that `cblaster` reads it in correctly. More help on building Entrez search queries can be found here.

Specifying filters

`cblaster` uses several filtering thresholds during the searching and clustering phases of its search workflow. These are listed below:

Argument	Description	Default
`-me/--max_evalue`	Max. E-value of a BLAST hit	0.01
`-mi/--min_identity`	Min. identity (%)	30
`-mc/--min_coverage`	Min. query coverage (%)	50
`-g/--gap`	Max. distance (bp) between any two hits in a cluster	20000
`-u/--unique`	Min. number of unique query sequences hit in a cluster	3
`-mh/--min_hits`	Min. number of total hits in a cluster	3
`-r/--require`	Query sequences that must be hit in a cluster	•

The default values for each filter are pretty generous, and may need changing based on your data. The search thresholds should be fairly self explanatory; any hit not meeting them are discarded from the BLAST search results.

The clustering thresholds, however, are a bit more interesting. These determine what conditions a candidate hit cluster must satisfy in order to be detected by `cblaster`. The most important argument here is `-g/--gap`, which determines how far (in base pairs) any two hits in a cluster can be from one another. This parameter could vary wildly based on your data set. For example, in bacterial or fungal secondary metabolite gene clusters where genes are typically found very close together, a low value could be used. Conversely, plant clusters, which may involve a collection of key genes spread out over the entire chromosome, would require a much higher value. The `gne` module can used to calibrate this parameter based on your results, and is described further in *Estimating genomic neighbourhood with the gne module*.

The `-u/--unique` and `-mh/--min_hits` arguments deal with the number of hits within candidate clusters. They differ in that `-u/--unique` looks for at least some number of your query sequences to be represented in given hit clusters, whereas `-mh/--min_hits` is only concerned with the total number of hits in the cluster, regardless of query sequence. For example, if I have five query sequences and I specify `-u 3`, any clusters that do not have hits corresponding to at least three of my query sequences will be discarded. However, if I have set `-mh 3`, any clusters that have less than three hits total in them will be discarded.

Finally, the `-r/--require` argument can be used to specify query sequences that must have hits in result clusters. Using the above example, we could specify three query sequences:

```
$ cblaster search -qf query.fasta -r Seq1 Seq3 Seq5
```

In this example, any clusters **not** containing Seq1, Seq3 and Seq5 will be discarded.

Specifying output

`cblaster` offers several useful output options for searches.

By default, a complete summary is generated and printed to the terminal after the search has finished. This reports all clusters, as well as the scores and positions of each gene hit, found during the search, organised by the organisms and genomic scaffolds they belong to. For example:

```
Pyricularia oryzae
==================
```

(continues on next page)

(continued from previous page)

```
CP034205.1
----------
Query  Subject      Identity  Coverage  E-value   Bitscore  Start    End      Strand
Seq1   QBZ57568.1   38.61     99.5235   0         2629      7879606  7891956  -
Seq2   QBZ57569.1   41.926    97.479    8.94e-90  285       7893739  7895354  -
Seq2   QBZ57572.1   32.979    98.324    3.97e-25  105       7900440  7901095  -
```

You can change how `cblaster` handles this output in several ways. To save this output to a file, you can use the `-o/--output` argument. The number of decimal places used in the score values can be changed using `-odc/--output_decimals`, and table headers can be hidden using `-ohh/--output_hide_headers`. You can also generate a character delimited summary (instead of human-readable) using the `-ode/--output_delimiter` argument. Throwing it all together, you could generate CSV file, with no headers and maximum 6 decimal places, and save it to a file like so:

```
$ cblaster search -qf query.fasta -o summary.csv -ode "," -ohh -odc 6
```

An easier way to digest all of the information that `cblaster` will produce is by using the binary table output. This generates a matrix which shows the absence/presence of query sequence (columns) hits in each result cluster (rows). For example:

```
Organism                            Scaffold         Start     End       BuaB   BuaC   BuaD␣
↪ BuaE
Aspergillus alliaceus CBS 536.65    NW_022474703.1   15435     43018     1      1      1    ␣
↪ 1
Aspergillus alliaceus CBS 536.65    NW_022474686.1   272633    304495    0      1      1    ␣
↪ 0
Aspergillus alliaceus IBT 14317     ML735331.1       15828     43603     1      1      1    ␣
↪ 1
Aspergillus alliaceus IBT 14317     ML735238.1       264335    296204    0      1      1    ␣
↪ 0
Aspergillus mulundensis DSM 5745    NW_020797889.1   1717881   1745289   1      1      1    ␣
↪ 1
Aspergillus versicolor IMB17-055    MN395477.1       2742      27898     1      1      1    ␣
↪ 1
Aspergillus versicolor CBS 583.65   KV878126.1       3162095   3187090   1      1      1    ␣
↪ 1
```

As with the regular output, you can save the binary table to a file, as well as hide headers, change decimal places and delimiters using their respective `-b/--binary` arguments:

```
$ cblaster search -qf query.fasta -b binary.csv -bde "," -bhh -bdc 6
```

By default, the binary table will only report the total number of hits per query sequence in each cluster. However, you can instead change this to some value calculated from the actual scores of hits in the clusters.

This is controlled by two additional arguments: `-bat/--binary_attribute`, which determines which score attribute ('identity', 'coverage', 'bitscore' or 'evalue') to use when calculating cell values, and `-bkey/--binary_key`, which determines the function ('len', 'max', 'sum') applied to the score attribute.

Each cell in the matrix refers to multiple hit sequences within each cluster. For every cell, the chosen score attribute is extracted from each hit corresponding to that cell. Then, the key function is applied to the extracted scores. The 'len' function calculates the length of each score list - essentially just counting the number of hits in that cell. The 'max' and 'sum' functions calculate the maximum and sum of each score list, respectively.

For example, given a cell:

```
Query: Seq1
Hits: Seq2 (50% identity), Seq3 (70% identity)
```

By default, the cell value would be 2 (i.e. the count of hits in the cluster for Seq1). You could instead get the maximum identity value in the cell:

```
$ cblaster search -qf query.fasta -b binary.txt -bat identity -bkey max
```

. . . which would report 0.7, or the sum of all identities in the cell:

```
$ cblaster search -qf query.fasta -b binary.txt -bat identity -bkey sum
```

. . . which would report 1.2.

`cblaster` is capable of producing rich, interactive visualisations based on the binary table using the `-p/--plot` argument. If no filename is provided to this argument, the plot will be served dynamically using Python's built in HTTP server, and you will have to terminate `cblaster` manually via an interrupt (usually Ctrl+C). If a filename is provided, `cblaster` will generate a static HTML file containing all of the necessary visualisation data and code, which can then be easily shared with other people.

Finally, `cblaster` allows you to save the raw BLAST and IPG tables downloaded from NCBI during a search, using the `--blast_file` and `--ipg_file` arguments, respectively.

Saving search sessions and recomputing outputs

Given that searches can take a significant time to run (i.e. as long as any normal batch BLAST job will take), `cblaster` is capable of saving a search session to file, and loading it back later for further filtering and visualisation. As mentioned above, to save a search session, use the `-s/--session` argument:

```
$ cblaster search -qf query.fasta -s session.json
```

Once the session is saved, any subsequent runs with that session specified will make `cblaster` try to load it instead of performing a new search. From here, you have a few cool options.

You can combine multiple session files (e.g. from local and remote searches) by providing more than one filename to the `-s/--session` argument:

```
$ cblaster search -s s1.json s2.json s3.json
[17:43:34] INFO - Loading session(s) [`s1.json', `s2.json', `s3.json']
...
```

Note: This requires each session file to correspond to the same query sequences; an error will be thrown if `cblaster` detects a mismatch.

You can recompute an old session using new filter thresholds to create a new session file:

```
$ cblaster search -s old.json -rcp new.json -g 40000 -mh 4
```

You can temporarily recompute (i.e. don't save) to generate a new visualisation:

```
$ cblaster search -s session.json -rcp -g 40000 -mh 4 -p plot.html
```

Note: Filtering this way is not destructive (i.e. does not modify the original file); all data is loaded, filtered and recomputed within the program itself.

Finding intermediate genes between hits

The default output for `cblaster` is the cluster heatmap, which shows the absence or presence of your query sequences. While we find this is generally the easiest way to pick up on patterns of cluster conservation, we also like to be able to visualise our results in their own genomic contexts so we can see the differences in gene order, orientation, size and so on.

For this reason, we added integration to the `clinker` tool (https://github.com/gamcil/clinker), which can generate highly interactive gene cluster comparison plots. However, in a regular `cblaster` search, we do not have access to any information about the genes **between** the BLAST hits shown in the heatmap. This means that if you were to run the `plot_clusters` module on your session file (see *Plotting extracted clusters using plot_clusters*), you would produce a figure where most of the clusters are missing genes!

To get around this, you can use the `-ig/--intermediate_genes` argument when performing a `cblaster` search. After the search has completed, genomic regions corresponding to the detected gene clusters are retrieved from the NCBI, and used to fill in the missing genes.

Note:

If you forgot to use `-ig/--intermediate_genes` during your search, don't fret. You can also use it on an existing session file alongside the `-rcp/--recompute` argument, to generate a new session file containing the missing genes. For example:

```
cblaster -s session.json -rcp new_session.json -ig
```

The intermediate genes feature has two other arguments:

Argument	Description	Default
`-md/--max_distance`	The maximum distance between the start/end of a cluster and an intermediate gene	5000bp
`--mic/ --maximum_clusters`	The maximum amount of clusters to find intermediate genes for	100

`-md/--max-distance` enables you to control how far `cblaster` will look for intermediate genes. By default, it is set to 5000bp, which covers the main cluster region (from the first hit to the last) plus some leeway on either side. Setting this to a higher value will allow for a broader analysis of the genome neighbourhood of each cluster.

`-mic/--maximum_clusters` controls how many clusters `cblaster` will attempt to find intermediate genes for. As each cluster has to be queried against the NCBI individually, this can take some time, so by default `cblaster` caps this at 100.

Creating local sequence databases with the `makedb` module

The `makedb` module is used to generate the databases used in local `cblaster` searches from genome files. `makedb` only takes two arguments: the genome files being used to build the databases, and some name to use when saving them. For example, generating a database from a set of genomes is as simple as:

```
$ cblaster makedb one.gbk two.gbk three.gbk four.gbk myDb
```

This will read in each GenBank file, then generate the files:

myDb.sqlite3	Local database used for looking up genomic context of hits
myDb.dmnd	DIAMOND sequence search database
myDb.fasta	All protein sequences parsed from genomes; used for HMMER searches

`cblaster` can also build databases from GFF3 files as above:

```
$ cblaster makedb one.gff two.gff three.gff four.gff myDb
```

In this case, `cblaster` will expect matching FASTA format files containing the nucleotide sequences for each sequence region in the corresponding GFF. For instance, in the above example, the working directory must also contain `one.fasta`, `two.fasta`, `three.fasta`, `four.fasta`.

Typically it is easiest to have all your genome files within a folder and use a wildcard to avoid having to type every file name, like so:

```
$ cblaster makedb genomes/*.gbk myDb
```

The shell will expand this automatically into a command that is functionally equivalent to the previous one. However, on Windows, we have run into some issues with this behaviour. For windows, instead use the command:

```
$ cblaster makedb (ls *.gbk | \% FullName) myDb
```

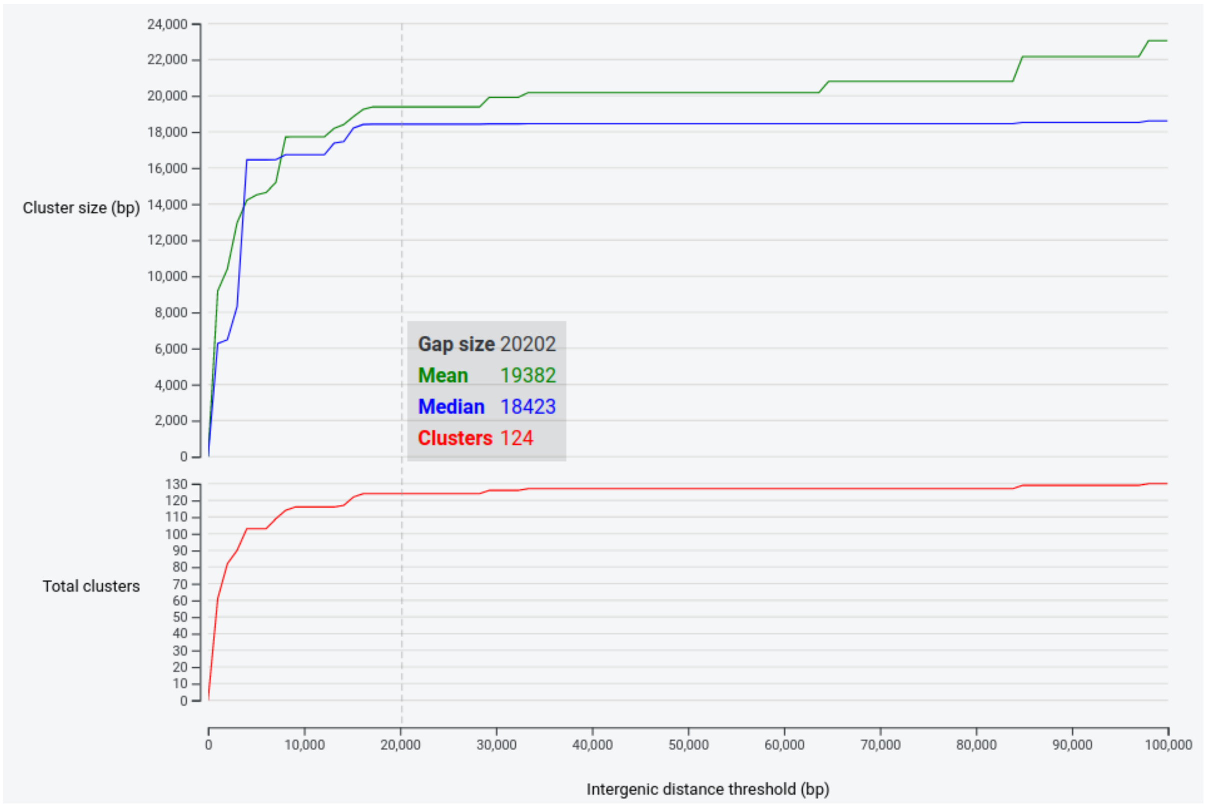
Estimating genomic neighbourhood with the `gne` module

In `cblaster`, the most important parameter when detecting hit clusters is the maximum inter-hit gap parameter. This determines how far `cblaster` will look between any two hits before terminating a given cluster. By default, this parameter is set to 20,000 bp; if no new hit is found within 20,000 bp of the previous hit in a cluster, `cblaster` will terminate extension of that cluster. Though the 20 kb cutoff has worked quite well for us when looking at fungi or bacteria, where gene density within clusters is quite high, it may not work for all datasets. For example, plant gene clusters may have key biosynthetic genes spread out over large stretches of the chromosome, with many genes in between; this is where the `gne` module comes in.

The `gne` module lets you robustly detect an appropriate value to use for this parameter by continually re-running cluster detection on a saved search session at different `gap` values over some interval. It then generates plots of the mean and median cluster sizes (bp), as well as the total number of predicted clusters, at each value. `gne` is run on a search sessions, like so:

```
$ cblaster gne session.json
```

And generate outputs that looks like this:



You can gain insight into the size of the given genomic neighbourhood of your query proteins – the result clusters in this case tend to be just under 20Kbp.

The `gne` module generates a list of `gap` values (total number determined by the `samples` parameter) from 0 to some upper limit (determined by the `max_gap` parameter). These numbers can be chosen in two ways. By default, `gne` will take evenly spaced (i.e. linear) values over the range 0-100,000 bp. Alternatively, you can choose to generate these values via a log scale, which will result in more samples at lower values than at higher ones. This can be specified using the `--scale` argument:

```
$ cblaster gne session.json --scale log
```

As these plots typically resemble logarithmic growth (i.e. rise steeply, then level off), it can make sense to sample more heavily in the more unstable region of the curve.

In case you would like the underlying data (e.g. for creating your own plots), `gne` can generate delimited output. To do this, simply use the `-o` or `--output` argument to specify a file to save the data to, and the `-d` or `--delimiter` argument to specify the delimiting character. For example, to generate a CSV file:

```
$ cblaster gne session.json --output gne.csv --delimiter ","
```

Like plots generated by the `search` module, `gne` plots can be saved as static HTML. To do this, provide a file to the `-p` or `--plot` argument:

```
$ cblaster gne session.json -p gne.html
```

Retrieving hit sequences with the `extract` module

After a search has been performed, it can be useful to retrieve sequences matching a certain query for further analyses (e.g. sequence comparisons for phylogenies). This is easily accomplished using `cblaster`'s `extract` module.

This module takes a cblaster session file as input, then extracts sequences matching any filters you have specified. If no filters are specified, ALL hit sequences will be extracted. However, that's probably not too useful, so instead we could extract all hit sequences matching a query sequence:

```
$ cblaster extract session.json -q "Query1"
```

By default, only sequence names are extracted. This is because cblaster stores no actual sequence data for hit sequences during it's normal search workflow, only their coordinates. However, sequences can automatically be retrieved from the NCBI by specifying the -d or --download argument. cblaster will then write them, in FASTA format, to either the command line or a file. For example, we can do the same command as above, but retrieve the sequences and write them to output.fasta like so:

```
$ cblaster extract session.json -q "Query1" -d -o output.fasta
```

Note that the -o or --output argument has been used here; this will write any results from the extract module to the specified file.

You can also provide multiple names of query sequences:

```
$ cblaster extract session.json -q Query1 Query2 Query3 ...
```

Note, however, that all extracted sequences will be written to the same file.

The extract module can also filter based on the organism or scaffold that each hit sequence is on. The organism filter uses regular expression patterns based on organism names. Multiple patterns can be provided, and are additive (i.e. any organism matching any of the patterns will be saved). For example, you could filter a search session of all fungal organisms on NCBI for only those sequences from *Aspergillus* or *Penicillium* species like so:

```
$ cblaster extract fungi.json -or "Aspergillus.*" "Penicillium.*"
```

Note that patterns should be enclosed in quotation marks in order to be read in correctly.

The scaffold filter is less flexible, capable of matching exact scaffolds or scaffold ranges. For example, to extract hit sequences on a scaffold, scaffold_1, from position 10000 to 23000:

```
$ cblaster extract session.json -sc "scaffold_1:10000-23000"
```

Like the organism filter, multiple scaffolds and/or scaffold ranges can be provided and they are additive.

By default, source information is added to each sequence name, for example:

```
sequence [organism=Source organism] [scaffold=scaffold_1:123-456]
```

This can be turned off using the -no or --names_only argument.

Finally, the *extract* module can also generate delimited table files, for easy importing into spreadsheet programs. For example, to generate a comma-delimited table (CSV file), simple provide the -de or --delimiter argument:

```
$ cblaster extract session.json ... -de ","
```

Extracting GenBank files from session files using `extract_clusters`

A common next step after a cblaster search is to retrieve the identified gene clusters so we can perform additional analysis. cblaster provides the extract_clusters module precisely for this purpose, allowing you to generate GenBank files of specific gene clusters directly from a session file. This works for sessions from both remote and local searches: for remote searches, clusters are downloaded directly from the NCBI, and in local searches, from the SQL database generated using the makedb module.

Example usage

Extract all clusters from a session (can take a long time for remote searches with many results):

```
$ cblaster extract_clusters session.json -o example_directory
```

Extract clusters 1-10 and cluster 25 (these numbers can be found in the summary file of the 'search' command):

```
$ cblaster extract_clusters session.json -c 1-10 25 -o example_directory
```

Extract clusters only from specific organisms (regular expressions):

```
$ cblaster extract_clusters session.json -or "Aspergillus.*" "Penicillium.*" -o␣
↪example_directory
```

Extract clusters only from a specific range on scaffold_123 and all clusters on scaffold_234 (note: expects unique scaffold names):

```
$ cblaster extract_clusters session.json -sc scaffold_123:1-80000 scaffold_234 -o␣
↪example_directory
```

Plotting extracted clusters using `plot_clusters`

By default, the visualisation offered by `cblaster` shows only a heatmap of query hits per result cluster. While this is very useful for quickly identifying patterns in large datasets, we generally still want to see how these clusters compare in a more biologically relevant way.

The `plot_clusters` module allows you to do precisely this. Given a session and some filters to choose specific clusters (exactly like in the `extract_clusters` module), this module will automatically extract the clusters, then generate an interactive visualisation showing each cluster to-scale using `clinker` (doi: 10.1093/bioinformatics/btab007, https://github.com/gamcil/clinker).

Example usage

Minimum working example:

```
$ cblaster plot_clusters session.json
```

Plot clusters 1-10 and cluster 25 (these numbers can be found in the summary file of the 'search' command):

```
$ cblaster plot_clusters session.json -c 1-10 25 -o plot.html
```

Plot only from specific organisms (regular expressions):

```
$ cblaster plot_clusters session.json -or "Aspergillus.*" "Penicillium.*" -o plot.html
```

Plot only clusters from a specific range on scaffold_123 and all clusters on scaffold_234 (note: assumes unique scaffold names):

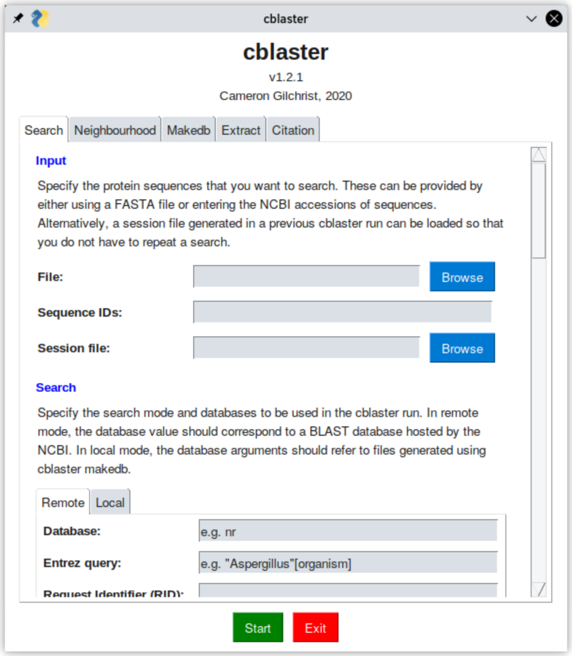
```
$ cblaster plot_clusters session.json -sc scaffold_123:1-80000 scaffold_234 -o plot.
↪html
```

Using the graphical user interface (GUI)

`cblaster` provides an easy to use graphical user interface (GUI) which is fully capable of performing all search functionality. The GUI is implemented using the `PySimpleGUI` Python package that is installed alongside `cblaster` automatically. To access the GUI, simply open a terminal and type:

```
$ cblaster gui
```

The GUI should then pop up in a new window. From there, you can run `cblaster` searches exactly as you would from the command line.



Input

Specify the protein sequences that you want to search. These can be provided by either using a FASTA file or entering the NCBI accessions of sequences. Alternatively, a session file generated in a previous cblaster run can be loaded so that you do not have to repeat a search.

File:

Sequence IDs:

Session file:

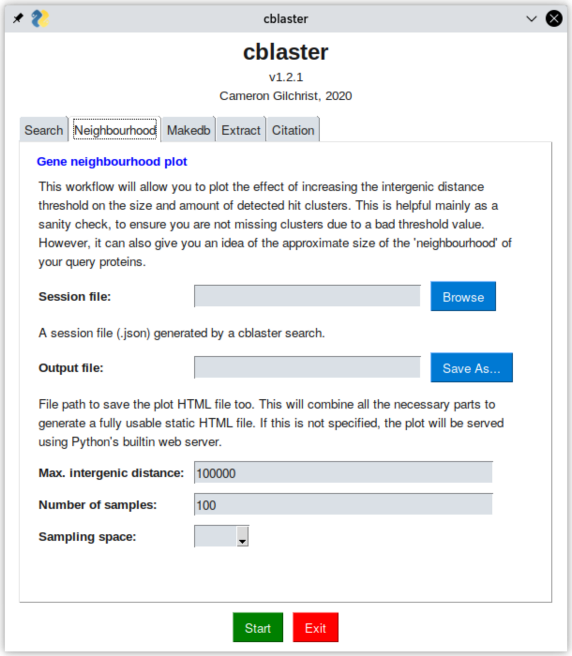
Search

Specify the search mode and databases to be used in the cblaster run. In remote mode, the database value should correspond to a BLAST database hosted by the NCBI. In local mode, the database arguments should refer to files generated using cblaster makedb.

Database: e.g. nr

Entrez query: e.g. "Aspergillus"[organism]

Request Identifier (RID):



Gene neighbourhood plot

This workflow will allow you to plot the effect of increasing the intergenic distance threshold on the size and amount of detected hit clusters. This is helpful mainly as a sanity check, to ensure you are not missing clusters due to a bad threshold value. However, it can also give you an idea of the approximate size of the 'neighbourhood' of your query proteins.

Session file:

A session file (.json) generated by a cblaster search.

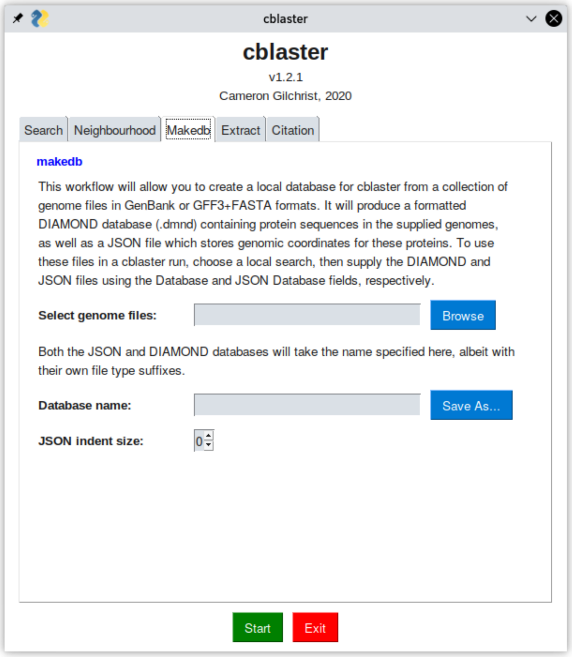
Output file:

File path to save the plot HTML file too. This will combine all the necessary parts to generate a fully usable static HTML file. If this is not specified, the plot will be served using Python's builtin web server.

Max. intergenic distance: 100000

Number of samples: 100

Sampling space:



makedb

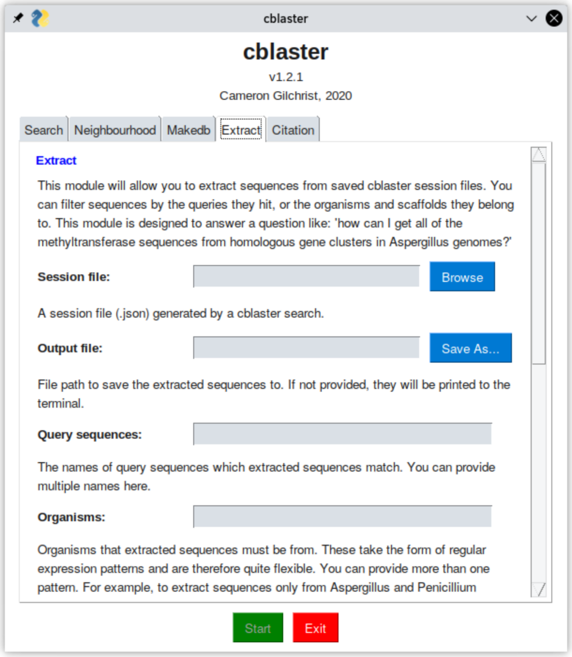
This workflow will allow you to create a local database for cblaster from a collection of genome files in GenBank or GFF3+FASTA formats. It will produce a formatted DIAMOND database (.dmnd) containing protein sequences in the supplied genomes, as well as a JSON file which stores genomic coordinates for these proteins. To use these files in a cblaster run, choose a local search, then supply the DIAMOND and JSON files using the Database and JSON Database fields, respectively.

Select genome files:

Both the JSON and DIAMOND databases will take the name specified here, albeit with their own file type suffixes.

Database name:

JSON indent size: 0



Extract

This module will allow you to extract sequences from saved cblaster session files. You can filter sequences by the queries they hit, or the organisms and scaffolds they belong to. This module is designed to answer a question like: 'how can I get all of the methyltransferase sequences from homologous gene clusters in Aspergillus genomes?'

Session file:

A session file (.json) generated by a cblaster search.

Output file:

File path to save the extracted sequences to. If not provided, they will be printed to the terminal.

Query sequences:

The names of query sequences which extracted sequences match. You can provide multiple names here.

Organisms:

Organisms that extracted sequences must be from. These take the form of regular expression patterns and are therefore quite flexible. You can provide more than one pattern. For example, to extract sequences only from Aspergillus and Penicillium

Miscellaneous functions

Accessing help dialogues

Every module in the `cblaster` command line interface has a useful help dialogue which details the arguments you may specify. To do this, simply type `-h/--help` as the sole argument after any command. For example:

```
$ cblaster -h
$ cblaster search -h
$ cblaster gne -h
$ cblaster makedb -h
$ cblaster extract -h
```

Getting the current `cblaster` version. To check the version of cblaster simply enter the command:

```
$ cblaster --version
```

JSON indent level

`cblaster` uses JSON files at several stages in its pipelines (i.e. search sessions, context databases). By default, these files are saved with no indentation level for the purpose of lowering file size. This means that all data is stored on a single line and is hard to read for anything but computers (e.g. human beings!). In most cases, this is perfectly fine; however, sometimes you may wish to investigate something manually within a search session and would require some level of indenting to make the file readable. This is achieved by using the `-i` or `--indent` argument. The `-i` argument belongs to the first level of the command line interface, and therefore must be used directly after `cblaster` in the command line string. So, this will **not** work:

```
$ cblaster search -qf query.fasta -s session.json -i 2
```

But **this will**:

```
$ cblaster -i 2 search -qf query.fasta -s session.json
```

This will set the indentation level of a given session to 2, meaning that for every new line in the file, 2 spaces will be drawn per indentation level. For example, a session file with no indent (truncated) looks like this:

```
{"queries": ["BuaB", "BuaC", "BuaD", "QBE85644.1", "BuaE", "BuaF", "BuaG",
"QBE85648.1", "BuaA"], "params": {"mode": "remote", "database": "nr",
"min_identity": 30, "min_coverage": 50, "max_evalue": 0.01, "query_file":
"bua.faa", "rid": "RAV3P2F3014"}, "organisms": [{"name": "Aspergillus sp.
CLMG-2019a", "strain": "FRR 5400", "scaffolds": ...
```

The session with indent 2 will look like this:

```
{
  "queries": [
    "BuaB",
    "BuaC",
    "BuaD",
    "QBE85644.1",
    "BuaE",
    "BuaF",
    "BuaG",
    "QBE85648.1",
```

(continues on next page)

(continued from previous page)

```
    "BuaA"
  ],
  "params": {
    "mode": "remote",
    "database": "nr",
    "min_identity": 30,
    "min_coverage": 50,
    "max_evalue": 0.01,
    "query_file": "bua.faa",
    "rid": "RAV3P2F3014"
  },
  "organisms": [
    {
        "name": "Aspergillus sp. CLMG-2019a",
        "strain": "FRR 5400",
        "scaffolds": [{ ... }]
        ...
```

Much more readable! Note though that, particularly in sessions with lots of results, this comes with a significant increase in file size.

CHAPTER 3

API Documentation

Comprehensive documentation for all public API exposed by *cblaster*:

3.1 API Documentation

3.1.1 cblaster.classes module

This module stores the classes (Organism, Scaffold, Hit) used in cblaster.

class `cblaster.classes.`**`Cluster`**(*indices=None*, *subjects=None*, *intermediate_genes=None*, *query_sequence_order=None*, *score=None*, *start=None*, *end=None*, *number=None*)

Bases: *`cblaster.classes.Serializer`*

A cluster of subjects on the same scaffold

`indices`

indexes of the subjects in the list of subjects

Type list

`of the parent scaffold`

`subjects`

Subject objects that are in this cluster. Note:

Type list

`These are not serialised for this cluster`

`intermediate_genes`

Type list

`start`

The start coordinate of the cluster on the parent scaffold

Type int

end
The end coordinate of the cluster on the parent scaffold

Type int

number
number that is unique for each cluster in order to identify them

Type int

NUMBER = count(0)

calculate_score(*query_sequence_order=None*)
Calculate the score of the current cluster

The score is based on accumulated blastbitscore, total amount of hits against the query and a synteny score if query sequence order is provided. If there are multiple hits in a subject the hit with the top bitscore is selected for the calculation.

Parameters

- **query_sequence_order** (*list*) – list of sequences of the order in the query file, is
- **provided if the query has a meningfull order** (*only*) –

Returns a float

classmethod from_dict(*d*, *subjects=None*)
Loads class from dict.

intermediate_end
The end of the cluster taking the intermediate genes into account

intermediate_start
The start of the cluster taking the intermediate genes into account

names

remove_subject(*subject*, *scaffold_index*)
Safely remove a subject from a cluster.

This is important when subjects become empty when recomputing a session with different treshholds

Parameters

- **subject** (Subject) – cblaster Subject object
- **scaffold_index** (*int*) – the index of the subject in the scaffold it is saved in

sequences

to_dict(*save_subjects=False*)
Serialises class to dict.

class cblaster.classes.**Hit**(*query*, *subject*, *identity*, *coverage*, *evalue*, *bitscore*)
Bases: *cblaster.classes.Serializer*

A BLAST hit identified during a cblaster search.

This class is first instantiated when parsing BLAST results, and is then updated with genomic coordinates after querying either the Identical Protein Groups (IPG) resource on NCBI, or a local JSON database.

query
Name of query sequence.

Type str

`subject`
Name of subject sequence.

Type str

`identity`
Percentage identity (%) of hit.

Type float

`coverage`
Query coverage (%) of hit.

Type float

`evalue`
E-value of hit.

Type float

`bitscore`
Bitscore of hit.

Type float

`copy`(***kwargs*)
Creates a copy of this Hit with any additional args.

`classmethod from_dict`(*d*)
Loads class from dict.

`to_dict`()
Serialises class to dict.

`values`(*decimals=4*)
Formats hit attributes for printing.

Parameters **`decimals`** (`int`) – Total decimal places to show in score values.

Returns List of formatted attribute strings.

`class` `cblaster.classes.`**`Organism`**(*name*, *strain*, *scaffolds=None*)
Bases: *`cblaster.classes.Serializer`*

A unique organism containing hits found in a cblaster search.

Every strain (or lack thereof) is a unique Organism, and will be reported separately in cblaster results.

`name`
Organism name, typically the genus and species epithet.

Type str

`strain`
Strain name of this organism, e.g. CBS 536.65.

Type str

`scaffolds`
Scaffold objects belonging to this organism.

Type dict

`clusters`

classmethod from_dict(*d*)
Loads class from dict.

full_name
The full name (including strain) of the organism. Note: if strain found in name, returns just name.

summary(*decimals=4*, *hide_headers=True*, *delimiter=None*)

to_dict()
Serialises class to dict.

total_hit_clusters
Counts total amount of hit clusters in this Organism.

class `cblaster.classes.`**Scaffold**(*accession*, *clusters=None*, *subjects=None*)
Bases: *`cblaster.classes.Serializer`*

A genomic scaffold containing hits found in a cblaster search.

accession
Name of this scaffold, typically NCBI accession.

Type str

subjects
Subject objects located on this scaffold.

Type list

clusters
Clusters of hits identified on this scaffold.

Type list

add_clusters(*subject_lists*, *query_sequence_order=None*)
Add clusters to this scaffold

After clusters are added they are sorted based on score

Parameters

- **subject_lists** (*`list`*) – a list of lists of Subject objects that are
- **a clusters** (*`form`*) –
- **query_sequence_order** (*`list`*) – list of sequences of the order in the query file, is
- **provided if the query has a meningfull order** (*`only`*) –

classmethod from_dict(*d*)
Loads class from dict.

remove_subject(*subject*)
Safely remove a subject from a cluster by removing it from the cluster as well.

Parameters subject (`Subject`) – cblaster Subject object

summary(*hide_headers=False*, *delimiter=None*, *decimals=4*)

to_dict()
Serialises class to dict.

class `cblaster.classes.`**Serializer**
Bases: `abc.ABC`

JSON serialisation mixin class.

Classes that inherit from this class should implement *to_dict* and *from_dict* methods.

classmethod from_dict(*d*)
Loads class from dict.

classmethod from_json(*js*)
Instantiates class from JSON handle.

to_dict()
Serialises class to dict.

to_json(*fp=None*, ***kwargs*)
Serialises class to JSON.

class cblaster.classes.**Session**(*queries=None*, *sequences=None*, *params=None*, *organisms=None*, *query=None*)
Bases: *cblaster.classes.Serializer*

Stores the state of a cblaster search.

This class stores query proteins, search parameters, Organism objects created during searches, as well as methods for generating summary tables. It can also be dumped to/loaded from JSON for re-filtering, plotting, etc.

```
>>> s = Session()
>>> with open("session.json", "w") as fp:
...     s.to_json(fp)
>>> with open("session.json") as fp:
...     s2 = Session.from_json(fp)
>>> s == s2
True
```

queries
Names of query sequences.

Type list

params
Search parameters.

Type dict

organisms
Organism objects created in a search.

Type list

sequences
Query sequence translations

Type dict

query
cblaster Cluster object for query

Type *Cluster*

format(*form*, *fp=None*, ***kwargs*)
Generates a summary table.

Parameters

- **form** (*str*) – Type of table to generate ('summary' or 'binary').
- **fp** (*file handle*) – File handle to write to.

Raises `ValueError` – *form* not 'binary' or 'summary'

Returns Summary table.

classmethod from_dict(*d*)
Loads class from dict.

classmethod from_file(*file*)

classmethod from_files(*files*)

to_dict()
Serialises class to dict.

class cblaster.classes.**Subject**(*id=None*, *hits=None*, *name=None*, *ipg=None*, *start=None*, *end=None*, *strand=None*, *sequence=None*)
Bases: *cblaster.classes.Serializer*

A sequence representing one or more BLAST hits.

This class is instantiated during the contextual lookup stage. It is important since it allows for subject sequences which hit >1 of the query sequences, while still staying non-redundant.

hits
Hit objects referencing this subject sequence.

Type list

ipg
NCBI Identical Protein Group (IPG) id.

Type int

start
Start of sequence on parent scaffold.

Type int

end
End of sequence on parent scaffold.

Type int

strand
Strandedness of the sequence ('+' or '-').

Type str

classmethod from_dict(*d*)
Loads class from dict.

to_dict()
Serialises class to dict.

values(*decimals=4*)

3.1.2 cblaster.context module

3.1.3 cblaster.database module

3.1.4 cblaster.extract module

3.1.5 cblaster.extract_clusters module

3.1.6 cblaster.formatters module

cblaster result formatters.

`cblaster.formatters.`**`add_field_whitespace`**(*rows*, *lengths*)

Fills table fields with whitespace to specified lengths.

`cblaster.formatters.`**`binary`**(*session*, *hide_headers=False*, *delimiter=None*, *key=<built-in function len>*, *attr='identity'*, *decimals=4*, *sort_clusters=False*)

Generates a binary summary table from a Session object.

`cblaster.formatters.`**`generate_header_string`**(*text*, *symbol='-'*)

Generates a 2-line header string with underlined text.

```
>>> header = generate_header_string("header string", symbol="*")
>>> print(header)
header string
*************
```

`cblaster.formatters.`**`get_cell_values`**(*queries*, *subjects*, *key=<built-in function len>*, *attr=None*)

Generates the values of cells in the binary matrix.

This function calls some specified key function (def. max) against all values of a specified attribute (def. None) from Hits inside Subjects which match each query. By default, this function will just count all matching Hits (i.e. len() is called on all Hits whose query attribute matches). To find maximum identities, for example, provide key=max and attr='identity' to this function.

Parameters

- **`queries`** (*`list`*) – Names of query sequences.
- **`subjects`** (*`list`*) – Subject objects to generate vlaues for.
- **`key`** (*`callable`*) – Some callable that takes a list and produces a value.
- **`attr`** (*`str`*) – A Hit attribute to calculate values with in key function.

`cblaster.formatters.`**`get_maximum_row_lengths`**(*rows*)

Finds the longest lengths of fields per column in a collection of rows.

`cblaster.formatters.`**`gne_summary`**(*data*, *hide_headers=False*, *delimiter=None*, *decimals=4*)

`cblaster.formatters.`**`humanise`**(*rows*)

Formats a collection of fields as human-readable.

`cblaster.formatters.`**`set_decimals`**(*value*, *decimals=4*)

`cblaster.formatters.`**`summarise_cluster`**(*cluster*, *decimals=4*, *hide_headers=True*, *delimiter=None*)

Generates a summary table for a hit cluster.

Parameters

- **cluster** (`Cluster`) – collection of Subject objects
- **decimals** (*int*) – number of decimal points to show
- **hide_headers** (*bool*) – hide column headers in output
- **delimiter** (*str*) – delimiting string between the subjects

Returns summary table

`cblaster.formatters.`**`summarise_gne`**(*data*, *hide_headers=False*, *delimiter=None*, *decimals=4*)

`cblaster.formatters.`**`summarise_organism`**(*organism*, *hide_headers=True*, *delimiter=None*, *decimals=4*)

`cblaster.formatters.`**`summarise_scaffold`**(*scaffold*, *hide_headers=True*, *delimiter=None*, *decimals=4*)

`cblaster.formatters.`**`summary`**(*session*, *hide_headers=False*, *delimiter=None*, *decimals=4*, *sort_clusters=False*)

3.1.7 cblaster.genome_parsers module

`cblaster.genome_parsers.`**`find_fasta`**(*gff_path*)
Finds a FASTA file corresponding to the given GFF path.

`cblaster.genome_parsers.`**`find_feature`**(*array*, *ftype*)

`cblaster.genome_parsers.`**`find_files`**(*paths*, *recurse=True*, *level=0*)

`cblaster.genome_parsers.`**`find_gene_name`**(*qualifiers*)
Finds a gene name in a dictionary of feature qualifiers.

`cblaster.genome_parsers.`**`find_overlapping_location`**(*feature*, *locations*)
Finds the index of a gene location containing *feature*.

Parameters

- **feature** (*SeqFeature*) – Feature being matched to a location
- **locations** (*list*) – Start and end coordinates of gene features

Returns Index of matching start/end, if any None: No match found

Return type int

`cblaster.genome_parsers.`**`find_regions`**(*directives*)
Looks for ##sequence-region directives in a list of GFF3 directives.

`cblaster.genome_parsers.`**`find_translation`**(*record*, *feature*)

`cblaster.genome_parsers.`**`iter_overlapping_features`**(*features*)

`cblaster.genome_parsers.`**`merge_cds_features`**(*features*)

`cblaster.genome_parsers.`**`organisms_to_tuples`**(*organisms*)
Generates insertion tuples from parsed organisms.

Parameters **organisms** (*list*) – Organism dictionaries parsed by *parse_file*

Returns SQLite3 database insertion tuples for all genes

Return type list

`cblaster.genome_parsers.`**`parse_cds_features`**(*features*, *record_start*) `cblaster.genome_parsers.`**`parse_file`**(*path*, *to_tuples=False*)

Dispatches a given file path to the correct parser given its extension.

Parameters

- **`path`** (*`str`*) – Path to genome file
- **`to_tuples`** (*`bool`*) – Generate insertion tuples from parsed SeqRecords

Returns File name and list of SeqRecord objects corresponding to scaffolds in file

Return type dict

`cblaster.genome_parsers.`**`parse_gff`**(*path*)

Parses GFF and corresponding FASTA using GFFutils.

Parameters **`path`** (*`str`*) – Path to GFF file. Should have a corresponding FASTA file of the same name with a valid FASTA suffix (.fa, .fasta, .fsa, .fna, .faa).

Returns SeqRecord objects corresponding to each scaffold in the file

Return type list

`cblaster.genome_parsers.`**`parse_infile`**(*path*, *format*)

`cblaster.genome_parsers.`**`return_file_handle`**(*input_file*)

Handles compressed and uncompressed files.

`cblaster.genome_parsers.`**`seqrecord_to_tuples`**(*record*, *source*)

3.1.8 cblaster.helpers module

`cblaster.helpers.`**`dict_to_cluster`**(*sequences*, *spacing=500*)

Creates a mock Cluster from a sequence dictionary.

`cblaster.helpers.`**`efetch_sequences`**(*headers*)

Retrieve protein sequences from NCBI for supplied accessions.

This function uses EFetch from the NCBI E-utilities to retrieve the sequences for all synthases specified in *headers*. The calls to EFetch can not exceed 500 accessions this means that the calls have to be limited. It then calls *fasta.parse* to parse the returned response; note that extra processing has to occur because the returned FASTA will contain a full sequence description in the header line after the accession.

Parameters **`headers`** (*`list`*) – Valid NCBI sequence identifiers (accession, GI, etc.).

Returns a dictionary of sequences keyed on header id

`cblaster.helpers.`**`fasta_seqrecords_to_cluster`**(*records*, *spacing=500*)

Creates a mock Cluster from a SeqIO FASTA parser handle.

`cblaster.helpers.`**`find_sqlite_db`**(*path*)

`cblaster.helpers.`**`form_command`**(*parameters*)

Flatten a dictionary to create a command list for use in subprocess.run()

`cblaster.helpers.`**`get_program_path`**(*aliases*)

Get programs path given a list of program names.

Parameters **`aliases`** (*`list`*) – Program aliases, e.g. [“diamond”, “diamond-aligner”]

Raises `ValueError` – Could not find any of the given aliases on system $PATH.

Returns Path to program executable.

`cblaster.helpers.`**`get_project_root`**()

`cblaster.helpers.`**`get_sequences`**(*query_file=None*, *query_ids=None*, *query_profiles=None*)

Convenience function to get dictionary of query sequences from file or IDs.

Parameters

- **`query_file`** (*`str`*) – Path to FASTA genbank or EMBL file containing query
- **`sequences.`** (*`protein`*) –
- **`query_ids`** (*`list`*) – NCBI sequence accessions.
- **`query_profiles`** (*`list`*) – Pfam profile accessions.

Raises `ValueError` – Did not receive values for query_file or query_ids.

Returns Dictionary of query sequences keyed on accession.

Return type sequences (dict)

`cblaster.helpers.`**`parse_query_sequences`**(*query_file=None*, *query_ids=None*, *query_profiles=None*)

Creates a Cluster object from query sequences.

If EMBL/GenBank, Cluster will use exact genomic coordinates parsed from file. Otherwise, a fake Cluster will be created where genes are drawn to scale, but always on positive strand and with fixed intergenic distance.

`cblaster.helpers.`**`seqrecord_to_cluster`**(*record*)

Creates a Cluster object from a SeqIO GenBank/EMBL parser handle.

`cblaster.helpers.`**`sequences_to_fasta`**(*sequences*)

Formats sequence dictionary as FASTA.

3.1.9 cblaster.hmm_search module

Hmmfetch and hmmsearch implementation

`cblaster.hmm_search.`**`check_pfam_db`**(*path*)

Check if Pfam-A db exists else download

Parameters **`path`** – String, path where to check

`cblaster.hmm_search.`**`fetch_pfam_profiles`**(*hmm*, *keys*)

Fetch hmm profiles from db and save in a file

Parameters

- **`db_path`** – String, path where db are stored
- **`keys_ls`** – String, Path to file with acc-nr

Returns List, strings with acc-numbers

Return type ls_keys

`cblaster.hmm_search.`**`get_pfam_accession`**(*dat_path: Union[str, pathlib.Path], keys: Collection[str]*) → Tuple[Set[str], Set[str]]

Get full accession number of Pfam profiles

Looks for keys in ID and AC attributes, such that accessions can be retrieved by name or accession.

Parameters

- **`keys`** – Strings of accession profiles numbers
- **`db_path`** – Path to dat.gz file with the full acc-nr

Returns List, string of full acc-number

Return type key_lines

`cblaster.hmm_search.`**`get_profile_names`**(*profiles: Collection[str]*) → Collection[str]

Extracts names from profile HMMs using regular expressions.

`cblaster.hmm_search.`**`group_profiles`**(*profiles: Collection[str]*) → tuple

Group input query profile HMMs by Pfam, custom or invalid.

If the profile is found on disk, it will be loaded directly. If not found locally, but starts with PF, will try to extract from Pfam. Otherwise, marked as invalid.

`cblaster.hmm_search.`**`parse_hmmer_output`**(*results*)

Parse hmmsearch output

Parameters **`file_list`** – List, string of file name of results that need parsing

Returns

list of class objects, with information

- query, subject, identity, coverage, e-value, bit score

Return type hit_info

`cblaster.hmm_search.`**`perform_hmmer`**(*fasta: str, query_profiles: List[str], pfam: str, session: cblaster.classes.Session, hmm_out: str = None*) → Optional[Collection[cblaster.classes.Hit]]

Main of running a hmmer search

Parameters

- **`fasta`** – Path to database FASTA file
- **`query_profiles`** – Pfam names/accessions, or paths to profile HMM files
- **`pfam`** – Path to folder containing Pfam database
- **`session`** – cblaster search session

Returns List of class objects with the hits

`cblaster.hmm_search.`**`read_profiles`**(*files: Collection[str]*) → Collection[str]

Reads in profile HMMs from a list of files.

`cblaster.hmm_search.`**`run_hmmsearch`**(*fasta*, *query*)

Run the hmmsearch command

Parameters

- **`path_pfam`** – String, Path to the pfam database
- **`path_db`** – String, Path to db that will be searched for profiles
- **`ls_keys`** – List, string of pfam profile names

Returns List, String of result file names

Return type temp_res

3.1.10 cblaster.intermediate_genes module

3.1.11 cblaster.local module

`cblaster.local.`**`diamond`**(*fasta*, *database*, *max_evalue=0.01*, *min_identity=30*, *min_coverage=50*, *hitlist_size=5000*, *cpus=None*, *sensitivity='fast'*)

Launch a local DIAMOND search against a database.

Parameters

- **`fasta`** (*`str`*) – Path to FASTA format query file
- **`database`** (*`str`*) – Path to DIAMOND database generated with cblaster makedb
- **`max_evalue`** (*`float`*) – Maximum e-value threshold
- **`min_identity`** (*`float`*) – Minimum identity (%) cutoff
- **`min_coverage`** (*`float`*) – Minimum coverage (%) cutoff
- **`hitlist_size`** (*`int`*) – Maximum number of hits to save
- **`cpus`** (*`int`*) – Number of CPU threads for DIAMOND to use

Returns Rows from DIAMOND search result table (split by newline)

Return type list

`cblaster.local.`**`parse`**(*results*, *min_identity=30*, *min_coverage=50*, *max_evalue=0.01*)

Parse a string containing results of a BLAST/DIAMOND search.

Parameters

- **`results`** (*`list`*) – Results returned by diamond() or blastp()
- **`min_identity`** (*`float`*) – Minimum identity (%) cutoff
- **`min_coverage`** (*`float`*) – Minimum coverage (%) cutoff
- **`max_evalue`** (*`float`*) – Maximum e-value threshold

Returns Hit objects representing hits that surpass scoring thresholds

Return type list

`cblaster.local.`**`search`**(*database*, *sequences=None*, *query_file=None*, *query_ids=None*, *blast_file=None*, *dmnd_sensitivity='fast'*, *min_identity=30*, *min_coverage=50*, *max_evalue=0.01*, *hitlist_size=5000*, ***kwargs*)

Launch a new BLAST search using either DIAMOND or command-line BLASTp (remote).

Parameters

- **`database`** (*`str`*) – Path to DIAMOND database
- **`sequences`** (*`dict`*) – Query sequences
- **`query_file`** (*`str`*) – Path to FASTA file containing query sequences
- **`query_ids`** (*`list`*) – NCBI sequence accessions
- **`blast_file`** (*`str`*) – Path to the file blast results are written to

Raises `ValueError` – No value given for query_file or query_ids

Returns Parsed rows with hits from DIAMOND results table

Return type list

3.1.12 cblaster.main module

3.1.13 cblaster.parsers module

Argument parsers.

`cblaster.parsers.`**`add_binary_arguments`**(*group*)

`cblaster.parsers.`**`add_clustering_group`**(*search*)

`cblaster.parsers.`**`add_config_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_extract_clusters_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_extract_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_filtering_group`**(*search*)

`cblaster.parsers.`**`add_gne_output_group`**(*parser*)

`cblaster.parsers.`**`add_gne_params_group`**(*parser*)

`cblaster.parsers.`**`add_gne_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_gui_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_input_group`**(*search*)

`cblaster.parsers.`**`add_intermediate_genes_group`**(*search*)

`cblaster.parsers.`**`add_makedb_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_output_arguments`**(*group*)

`cblaster.parsers.`**`add_output_group`**(*search*)

`cblaster.parsers.`**`add_plot_clusters_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_search_subparser`**(*subparsers*)

`cblaster.parsers.`**`add_searching_group`**(*search*)

`cblaster.parsers.`**`full_database_path`**(*database*, **acces_modes*)

Make sure the database path is also correct, but do not check when providing one of the NCBI databases

Parameters

- **`database`** (`str`) – a string that is the path to the database creation files or a NCBI database identifier
- **`acces_modes`** (`List`) – a list of integers of acces modes for which at least one should be allowed

Returns a string that is the full path to the database file or a NCBI database identifier

`cblaster.parsers.`**`full_path`**(*file_path*, **acces_modes*, *dir=False*)

Test if a file path or directory exists and has the correct permissions and create a full path

For reading acces the file has to be pressent and there has to be read acces. For writing acces the directory with the file has to be present and there has to be write acces in that directory.

Parameters

- **`file_path`** (`str`) – relative or absoluete path to a file
- **`acces_modes`** (`List`) – a list of integers of acces modes for which at least one should be allowed

- **dir** (*bool*) – if the path is to a directory or not

Returns A string that is the full path to the provided file_path

Raises

- argparse.ArgumentTypeError when the provided path does not exist or the file does not have the correct
- permissions to be accessed

`cblaster.parsers.`**`get_parser`**`()`

`cblaster.parsers.`**`max_cpus`**`(`*value*`)`

Ensure that the cpu's do not go above the available amount. Setting to high cpu's will crash database creation badly

Parameters **value** (*int*) – number of cpu's as provided by the user

Returns value as an integer with 1 <= value <= multiprocessing.cpu_count()

`cblaster.parsers.`**`parse_args`**`(`*args*`)`

3.1.14 cblaster.plot module

3.1.15 cblaster.plot_clusters module

3.1.16 cblaster.remote module

This module handles all interaction with NCBI's BLAST API, including launching new remote searches, polling for completion status, and retrieval of results.

`cblaster.remote.`**`check`**`(`*rid*`)`

Check completion status of a BLAST search given a Request Identifier (RID).

Parameters **rid** (*str*) – NCBI BLAST search request identifier (RID)

Returns Search has completed successfully and hits were reported

Return type bool

Raises

- `ValueError` – Search has failed. This is caused either by program error (in which case, NCBI requests you submit an error report with the RID) or expiration of the RID (only stored for 24 hours).
- `ValueError` – Search has completed successfully, but no hits were reported.

`cblaster.remote.`**`parse`**`(`*handle*, *sequences=None*, *query_file=None*, *query_ids=None*, *max_evalue=0.01*, *min_identity=30*, *min_coverage=50*`)`

Parse Tabular results from remote BLAST search performed via API.

Since the API provides no option for returning query coverage, which is a metric we want to use for filtering hits, query sequences must be passed to this function so that their lengths can be compared to the alignment length.

Parameters

- **handle** (*list*) – File handle (or file handle-like) object corresponding to BLAST results. Note that this function expects an iterable of tab-delimited lines and performs no validation/error checking

- **sequences** (*dict*) – Query sequences
- **query_file** (*str*) – Path to FASTA format query file
- **query_ids** (*list*) – NCBI sequence identifiers
- **max_evalue** (*float*) – Maximum e-value
- **min_identity** (*float*) – Minimum percent identity
- **min_coverage** (*float*) – Minimum percent query coverage

Returns Hit objects corresponding to criteria passing BLAST hits

Return type list

`cblaster.remote.`**`poll`**(*rid*, *delay=60*, *max_retries=-1*)

Poll BLAST API with given Request Identifier (RID) until results are returned.

As per NCBI usage guidelines, this function will only poll once per minute; this is calculated each time such that wait is constant (i.e. accounts for differing response time on the status check).

Parameters

- **rid** (*str*) – NCBI BLAST search request identifier (RID)
- **delay** (*int*) – Total delay (seconds) between polling
- **max_retries** (*int*) – Maximum number of polling attempts (-1 for unlimited)

Returns BLAST search results split by newline

Return type list

`cblaster.remote.`**`retrieve`**(*rid*, *hitlist_size=5000*)

Retrieve BLAST results corresponding to a given Request Identifier (RID).

Parameters

- **rid** (*str*) – NCBI BLAST search request identifiers (RID)
- **hitlist_size** (*int*) – Total number of hits to retrieve

Returns BLAST search results split by newline, with HTML parts removed

Return type list

`cblaster.remote.`**`search`**(*rid=None*, *sequences=None*, *query_file=None*, *query_ids=None*, *min_identity=0.3*, *min_coverage=0.5*, *max_evalue=0.01*, *blast_file=None*, *hitlist_size=500*, ***kwargs*)

Perform a remote BLAST search via the NCBI's BLAST API.

This function launches a new search given a query FASTA file or list of valid NCBI identifiers, polls the API to check the completion status of the search, then retrieves and parses the results.

It is also possible to call other BLAST variants using the program argument.

Parameters

- **rid** (*str*) – NCBI BLAST search request identifier (RID)
- **sequences** (*dict*) – Query sequences
- **query_file** (*str*) – Path to FASTA format query file
- **query_ids** (*list*) – NCBI sequence identifiers
- **min_identity** (*float*) – Minimum percent identity
- **min_coverage** (*float*) – Minimum percent query coverage

- **max_evalue** (*float*) – Maximum e-value
- **blast_file** (*str*) – Path to file blast results are written to
- **hitlist_size** (*int*) – Number of database sequences to keep

Returns Hit objects corresponding to criteria passing BLAST hits

Return type list

`cblaster.remote.start`(*sequences=None, query_file=None, query_ids=None, database='nr', program='blastp', megablast=False, filtering='F', evalue=0.1, nucl_reward=None, nucl_penalty=None, gap_costs='11 1', matrix='BLOSUM62', hitlist_size=500, threshold=11, word_size=6, comp_based_stats=2, entrez_query=None*)

Launch a remote BLAST search using NCBI BLAST API.

Note that the HITLIST_SIZE, ALIGNMENTS and DESCRIPTIONS parameters must all be set together in order to mimic max_target_seqs behaviour.

Usage guidelines:

1. Don't contact server more than once every 10 seconds
2. Don't poll for a single RID more than once a minute
3. Use URL parameter email/tool
4. Run scripts weekends or 9pm-5am Eastern time on weekdays if >50 searches

For a full description of the parameters, see:

1. *BLAST API documentation<https://ncbi.github.io/blast-cloud/dev/api.html>*
2. *BLAST documentation <https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=Blas*

Parameters

- **sequences** (*dict*) – Query sequence dict generated by helpers.get_sequences()
- **query_file** (*str*) – Path to a query FASTA file
- **query_ids** (*list*) – Collection of NCBI sequence identifiers
- **database** (*str*) – Target NCBI BLAST database
- **program** (*str*) – BLAST variant to run
- **megablast** (*bool*) – Enable megaBLAST option (only with BLASTn)
- **filtering** (*str*) – Low complexity filtering
- **evalue** (*float*) – E-value cutoff
- **nucl_reward** (*int*) – Reward for matching bases (only with BLASTN/megaBLAST)
- **nucl_penalty** (*int*) – Penalty for mismatched bases (only with BLASTN/megaBLAST)
- **gap_costs** (*str*) – Gap existence and extension costs
- **matrix** (*str*) – Scoring matrix name
- **hitlist_size** (*int*) – Number of database sequences to keep
- **threshold** (*int*) – Neighbouring score for initial words
- **word_size** (*int*) – Size of word for initial matches

- **`comp_based_stats`** (*int*) – Composition based statistics algorithm
- **`entrez_query`** (*str*) – NCBI Entrez search term for pre-filtering the BLAST database

Returns Request Identifier (RID) assigned to the search rtoe (int): Request Time Of Execution (RTOE), estimated run time of the search

Return type rid (str)

3.1.17 cblaster.sql module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index