pyhmmer

Release 0.3.1

Martin Larralde

May 09, 2021

CONTENTS

Cython bindings and Python interface to HMMER3.

CHAPTER

ONE

OVERVIEW

HMMER is a biological sequence analysis tool that uses profile hidden Markov models to search for sequence homologs. HMMER3 is maintained by members of the the Eddy/Rivas Laboratory at Harvard University.

`pyhmmer` is a Python module, implemented using the Cython language, that provides bindings to HMMER3. It directly interacts with the HMMER internals, which has the following advantages over CLI wrappers:

- **single dependency**: If your software or your analysis pipeline is distributed as a Python package, you can add *`pyhmmer`* as a dependency to your project, and stop worrying about the HMMER binaries being properly setup on the end-user machine.
- **no intermediate files**: Everything happens in memory, in Python objects you have control on, making it easier to pass your inputs to HMMER without needing to write them to a temporary file. Output retrieval is also done in memory, via instances of the *`pyhmmer.plan7.TopHits`* class.
- **no input formatting**: The Easel object model is exposed in the *`pyhmmer.easel`* module, and you have the possibility to build a *`Sequence`* object yourself to pass to the HMMER pipeline. This is useful if your sequences are already loaded in memory, for instance because you obtained them from another Python library (such as Pyrodigal or Biopython).
- **no output formatting**: HMMER3 is notorious for its numerous output files and its fixed-width tabular output, which is hard to parse (even `Bio.SearchIO.HmmerIO` is struggling on some sequences).
- **efficient**: Using *`pyhmmer`* to launch `hmmsearch` on sequences and HMMs in disk storage is typically faster than directly using the `hmmsearch` binary. *`pyhmmer.hmmer.hmmsearch`* uses a different parallelisation strategy compared to the `hmmsearch` binary from HMMER, which helps getting the most of multiple CPUs.

CHAPTER

TWO

SETUP

Run `pip install pyhmmer` in a shell to download the latest release and all its dependencies from PyPi, or have a look at the *Installation page* to find other ways to install `pyhmmer`.

CHAPTER

THREE

LIBRARY

3.1 Installation

Note: Wheels are provided for Linux x86-64 platforms, but other machines will have to build the wheel from the source distribution. Building `pyhmmer` involves compiling HMMER3 and Easel, which requires a C compiler to be available.

3.1.1 PyPi

`pyhmmer` is hosted on GitHub, but the easiest way to install it is to download the latest release from its PyPi repository. It will install all dependencies then install `pyhmmer` either from a wheel if one is available, or from source after compiling the Cython code :

```
$ pip install --user pyhmmer
```

3.1.2 EMBL Package Registry

You can also install `manylinux` wheels built from the latest commit that passed the unit tests. Those bleeding-edge releases are available in the GitLab Package Registry hosted on the EMBL `git` server. Just instruct `pip` to use an extra index URL as follow:

```
$ pip install --user pyhmmer --extra-index-url https://git.embl.de/api/v4/projects/
↪3638/packages/pypi/simple
```

3.1.3 GitHub + `pip`

If, for any reason, you prefer to download the library from GitHub, you can clone the repository and install the repository by running (with the admin rights):

```
$ pip install --user https://github.com/althonos/pyhmmer/archive/master.zip
```

Caution: Keep in mind this will install always try to install the latest commit, which may not even build, so consider using a versioned release instead.

3.1.4 GitHub + `setuptools`

If you do not want to use `pip`, you can still clone the repository and run the `setup.py` file manually, although you will need to install the build dependencies (mainly Cython):

```
$ git clone --recursive https://github.com/althonos/pyhmmer
$ cd pyhmmer
$ python setup.py build
# python setup.py install
```

Danger: Installing packages without `pip` is strongly discouraged, as they can only be uninstalled manually, and may damage your system.

3.2 Examples

3.2.1 Multiple Sequence Alignment to HMM

[1]:
```
import pyhmmer
pyhmmer.__version__
```

[1]:
```
'0.3.1'
```

[2]:
```
alphabet = pyhmmer.easel.Alphabet.amino()
```

Loading the alignment

A new HMM can be built from a single sequence, or from a multiple sequence alignment. Let's load an alignment in digital mode so that we can build our HMM:

[3]:
```
with pyhmmer.easel.MSAFile("data/msa/LuxC.sto") as msa_file:
    msa_file.set_digital(alphabet)
    msa = next(msa_file)
```

Note

In this example, we load a multiple sequence alignment from a file, but if your program produces alignment and you wish to produce an HMM out of them, you can instantiate a `DigitalMSA` object yourself, e.g.:

```
seq1 = pyhmmer.easel.TextSequence(name="seq1", sequence="WVPKQDFT")
seq2 = pyhmmer.easel.TextSequence(name="seq2", sequence="WL--PQGE")
msa  = pyhmmer.easel.DigitalMSA(name="msa", sequences=[seq1, seq2])
```

Because we need a `DigitalMSA` to build the HMM, you will have to convert it first:

```
msa_d = msa.digitize(alphabet)
```

Building an HMM

Now that we have a multiple alignment loaded in memory, we can build a pHMM using a `pyhmmer.plan7.Builder`. This also requires a Plan7 background model to compute the transition probabilities.

[4]:
```
builder = pyhmmer.plan7.Builder(alphabet)
background = pyhmmer.plan7.Background(alphabet)
hmm, _, _ = builder.build_msa(msa, background)
```

We can have a look at the consensus sequence of the HMM with the `consensus` property:

[5]:
```
hmm.consensus
```

[5]:
```
↪'lanlkleeildlleevaqrlkdeeysrryirelakilgyeeemlkalkalmallskeaLkdllereLgqpeildeflprkesyekaqpkglvvhllaq
↪'
```

Saving the resulting HMM

Now that we have an HMM, we can save it to a file to avoid having to rebuild it every time. Using the `HMM.write` method lets us write the HMM in ASCII format to an arbitrary file. The resulting file will also be compatible with the `hmmsearch` binary if you wish to use that one instead of PyHMMER.

[6]:
```
with open("data/hmms/txt/LuxC.hmm", "wb") as output_file:
    hmm.write(output_file)
```

Applying the HMM to a sequence database

Once a pHMM has been obtained, it can be applied to a sequence database with the `pyhmmer.plan7.Pipeline` object. Let's iterate over the protein sequences in a FASTA to see if our new HMM gets any hits:

[7]:
```
pipeline = pyhmmer.plan7.Pipeline(alphabet, background=background, report_e=1e-5)

with pyhmmer.easel.SequenceFile("data/seqs/LuxC.faa") as seq_file:
    seq_file.set_digital(alphabet)
    hits = pipeline.search_hmm(query=hmm, sequences=seq_file)
```

We can then query the `TopHits` object to access the domain hits in the sequences:

[8]:
```
ali = hits[0].domains[0].alignment

print(" "*3, ali.target_name.decode())
print("{:3}".format(ali.hmm_from), ali.hmm_sequence[:80] + "...")
print(" "*3, ali.identity_sequence[:80] + "...")
print("{:3}".format(ali.target_from), ali.target_sequence[:80] + "...")
print(" "*3, ali.hmm_name.decode())
```

```
    tr|B6ESM7|B6ESM7_ALISL
  2 anlkleeildlleevaqrlkdeeysrr..
↪yirelakilgyeeemlkalka...lmallskeaLkdllereLgqpeildef...
    +nl+l++++++l++v+qr+++eey+rr  yir+l+++lgy++em+k l+a   +m l+sk+aL+d+++++Lg+␣
↪+i+de+...
 50 NNLRLNQVVNFLYTVGQRWRSEEYTRRrtYIRDLTNFLGYSNEMAK-
↪LEAnwiAMLLCSKSALYDIVQHDLGSLHIIDEW...
    LuxC
```

3.2.2 Active Site Analysis

This example is adapted from the method used by AntiSMASH to annotate biosynthetic gene clusters. AntiSMASH uses profile HMMs to annotate enzymatic domains in protein sequences. By matching the amino acids in the alignment, it can then predict the product specificity of the enzyme.

In this notebook, we show how to reproduce this kind of analysis, using a PKSI Acyltransferase domain built by the AntiSMASH authors (the HMM in HMMER2 format can be downloaded from their git repository).

References

- Del Vecchio, F., H. Petkovic, S. G. Kendrew, L. Low, B. Wilkinson, R. Lill, J. Cortes, B. A. Rudd, J. Staunton, and P. F. Leadlay. 2003. *Active-site residue, domain and module swaps in modular polyketide synthases.* J Ind. Microbiol Biotechnol 30:489-494.
- Medema MH, Blin K, Cimermancic P, de Jager V, Zakrzewski P, Fischbach MA, Weber T, Takano E, Breitling R. *antiSMASH: rapid identification, annotation and analysis of secondary metabolite biosynthesis gene clusters in bacterial and fungal genome sequences.* Nucleic Acids Res. 2011 Jul:W339-46.

[1]:
```
import pyhmmer
pyhmmer.__version__
```

[1]:
```
'0.3.1'
```

Loading the HMM

Loading a HMMER profile is done with the `pyhmmer.plan7.HMMFile` class, which provides an iterator over the HMMs in the file. Since we only use a single HMM, we can simply use `next` to get the first (and only) `pyhmmer.plan7.HMM`.

[2]:
```
with pyhmmer.plan7.HMMFile("data/hmms/txt/PKSI-AT.hmm") as hmm_file:
    hmm = next(hmm_file)
```

Building digitized sequences

Easel provides the code necessary to load sequences from files in common biological formats, such as GenBank or FASTA. These utilities are wrapped by the `pyhmmer.easel.SequenceFile`, which provides an iterator over the sequences in the file. Note that `SequenceFile` tries to guess the format by default, but you can force a particular format with the `format` keyword argument.

[3]:
```
with pyhmmer.easel.SequenceFile("data/seqs/PKSI.faa") as seq_file:
    seq_file.set_digital(hmm.alphabet)
    sequences = list(seq_file)
```

Note

The C interface of Easel allows storing a sequence in two different modes: in *text* mode, where the sequence letters are represented as individual characters (e.g. "A" or "Y"), and *digital* mode, where sequence letters are encoded as digits. To make Python programs clearer, and to allow static typecheck of the storage mode, we provide two separate classes, `TextSequence` and `DigitalSequence`, that represent a sequence stored in either of these modes.

`SequenceFile` yields sequences in text mode, but HMMER expects sequences in digital mode, so we must digitize them. This requires the sequence alphabet to be known, but we can just use the `Alphabet` instance stored in the `alphabet` attribute of `hmm`.

Running a search pipeline

With the sequences and the HMM ready, we can finally run the search pipeline: it has to be initialized with an `Alphabet` instance, so that the Plan7 background model can be configured accordingly. Then, we run the pipeline in search mode, providing it one HMM, and several sequences. This method returns a `TopHits` instance that is already sorted and thresholded.

[4]:
```
pipeline = pyhmmer.plan7.Pipeline(hmm.alphabet)
hits = pipeline.search_hmm(hmm, sequences)
```

Rendering the alignments

`Domain` instances store all the required information to report results in their `alignment` attribute. We can show the alignment between a HMM and a sequence like `hmmsearch` would as follow (using the first domain of the first hit as an example):

[5]:
```
ali = hits[0].domains[0].alignment

print(" "*3, ali.target_name.decode())
print("{:3}".format(ali.hmm_from), ali.hmm_sequence, "{:3}".format(ali.hmm_to))
print(" "*3, ali.identity_sequence)
print("{:3}".format(ali.target_from), ali.target_sequence, "{:3}".format(ali.target_
↪to))
print(" "*3, ali.hmm_name.decode())
```

```
    sp|Q9ZGI5|PIKA1_STRVZ
  1␣
↪lFpGQGsQyaGMGreLYetePVFRqalDrCaaaLrphLgfsLlevLfgdegqeeaaaslLdqTryaQPALFAvEYALArLWrSWGvePdAVlGHSvGEy
↪lpggGaMlaVraseeevrelLapyggrlsiAAvNGPrsvVvSGdaeaieallaeLeaqGirarrLkVsHAFHSplMepmldeleevlagitpraPriPl
↪308
    +FpGQG+Q+aGMG eL++++ VF++a+ +C+aaL+p++++sL +v ++ +g     a+ L++++++QP+ FAv+++LAr␣
↪W+  Gv+P+AV+GHS+GE++AA+vAG+lSL+DA+r+V  R++ ++a l+g+G+Ml+ ++se+ v e+La+++ +ls+AAvNGP␣
↪++VvSGd+ +ie+l++++ea G+rar ++V++A+HS+++e +  el+evlag++p+aPr+P++S++ G+w+t+ ␣
↪+ld++YW+r+lR+ V Fa+++etL+ + G+t+F+Ev++hpvLt ++  t            + la+Lrr+
635 VFPGQGTQWAGMGAELLDSSAVFAAAMAECEAALSPYVDWSLEAVVRQAPG-----
↪APTLERVDVVQPVTFAVMVSLARVWQHHGVTPQAVVGHSQGEIAAAYVAGALSLDDAARVVTLRSKSIAAhLAGKGGMLSLALSEDAVLERLAGFD-
↪GLSVAAVNGPTATVVSGDPVQIEELARACEADGVRARVIPVDYASHSRQVEIIESELAEVLAGLSPQAPRVPFFSTLEGAWITE-
↪PVLDGGYWYRNLRHRVGFAPAVETLATDEGFTHFVEVSAHPVLTMALPGTV----------TGLATLRRD 925
    PKS-AT.tcoffee
```

You may also want to see where the domains are located in the input sequence; using the DNA feature viewer developed by the Edinburgh Genome Foundry, we can build a summary graph aligning the protein sequences to the same reference axis:

[6]:
```
from dna_features_viewer import GraphicFeature, GraphicRecord
import matplotlib.pyplot as plt

# create an index so we can retrieve a Sequence from its name
seq_index = { seq.name:seq for seq in sequences }

fig, axes = plt.subplots(nrows=len(hits), figsize=(16, 6), sharex=True)
```

(continues on next page)

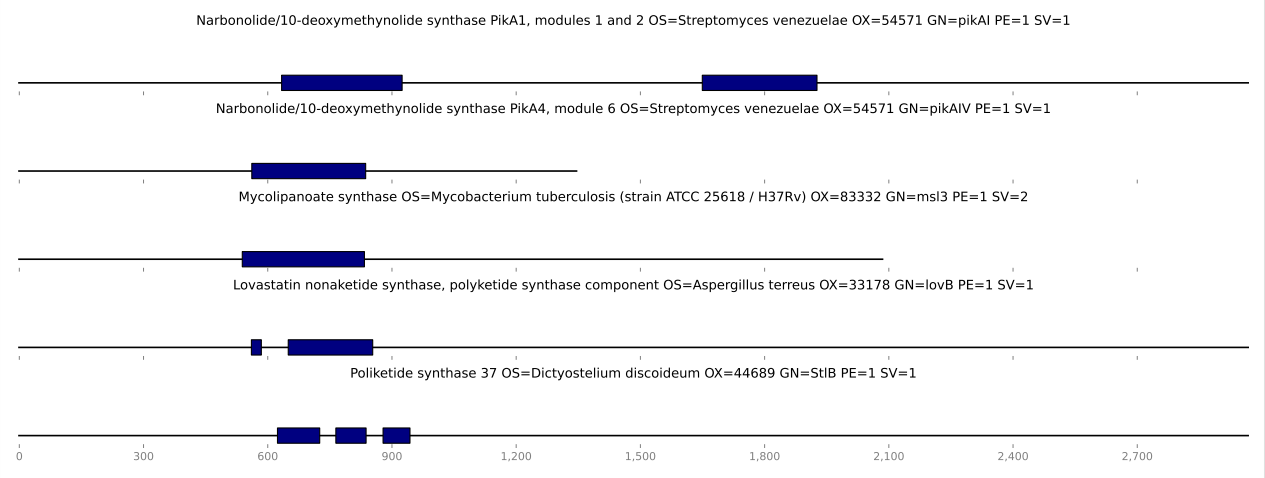
(continued from previous page)

```
for ax, hit in zip(axes, hits):
    # add one feature per domain
    features = [
        GraphicFeature(start=d.alignment.target_from-1, end=d.alignment.target_to)
        for d in hit.domains
    ]
    length = len(seq_index[hit.name])
    desc = seq_index[hit.name].description.decode()

    # render the feature records
    record = GraphicRecord(sequence_length=length, features=features)
    record.plot(ax=ax)
    ax.set_title(desc)

# make sure everything fits in the final graph!
fig.tight_layout()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/pyhmmer/envs/v0.3.1/lib/python3.7/
↪site-packages/traitlets/traitlets.py:3036: FutureWarning: --rc={'figure.dpi': 96}␣
↪for dict-traits is deprecated in traitlets 5.0. You can pass --rc <key=value> ...␣
↪multiple times to add items to a dict.
  FutureWarning,
```

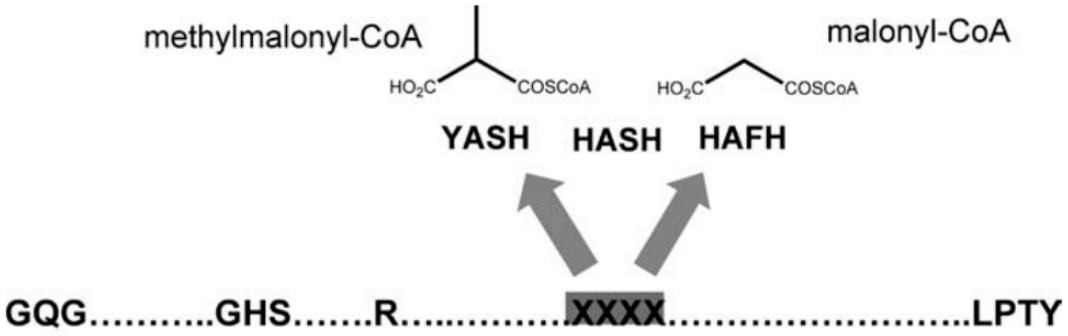


Checking individual positions for catalytic activity

First let's define a function to iterate over an alignement; this will come in handy later. This function yields the position in the alignment (using the HMM coordinates) and the aligned amino acid, skipping over gaps in the HMM sequence.

[7]:
```
def iter_target_match(alignment):
    position = alignment.hmm_from
    for hmm_letter, amino_acid in zip(alignment.hmm_sequence, alignment.target_
↪sequence):
        if hmm_letter != ".":
            yield position, amino_acid
            position += 1
```

Now, for the final step, we want to check for the specificity of the enzyme domains; Del Vecchio *et al.* have identified two amino acids in the acyltransferase domain that once muted will decide of the enzyme specificity for either malonyl-CoA or methylmalonyl-CoA:



For this, we need to check the alignment produced by HMMER, and verify the residues of the catalytic site correspond to the ones expected by the authors. We use the function we defined previously, first to check the core amino acids are not muted, and then to check the specificity of the two remaining residues.

[8]:
```
POSITIONS   = [ 93,  94,  95, 120, 196, 198]
EXPECTED    = ['G', 'H', 'S', 'R', 'A', 'H']
SPECIFICITY = [195, 197]

for hit in hits:
    print("\nIn sequence {!r}:".format(hit.name.decode()))
    for domain in hit.domains:
        ali = domain.alignment
        aligned = dict(iter_target_match(ali))

        print("- Found PKSI-AT domain at positions {:4} to {:4}".format(ali.target_
↪from, ali.target_to))
        try:
            signature = [ aligned[x] for x in POSITIONS ]
            spec = [ aligned[x] for x in SPECIFICITY ]
        except KeyError:
            print("  -> Domain likely too short")
            continue
        if signature != EXPECTED:
            print("  -> Substrate specificity unknown")
        elif spec == ["H", "F"]:
            print("  -> Malonyl-CoA specific")
        elif spec == ["Y", "S"]:
            print("  -> Methylmalonyl-CoA specific")
        else:
            print("  -> Neither malonyl-CoA nor methylmalonyl-CoA specific")
```

```
In sequence 'sp|Q9ZGI5|PIKA1_STRVZ':
- Found PKSI-AT domain at positions  635 to  925
  -> Methylmalonyl-CoA specific
- Found PKSI-AT domain at positions 1651 to 1927
  -> Methylmalonyl-CoA specific
- Found PKSI-AT domain at positions 3181 to 3475
  -> Malonyl-CoA specific

In sequence 'sp|Q9ZGI2|PIKA4_STRVZ':
- Found PKSI-AT domain at positions  563 to  837
  -> Methylmalonyl-CoA specific
```

(continues on next page)

(continued from previous page)

```
In sequence 'sp|A0A089QRB9|MSL3_MYCTU':
- Found PKSI-AT domain at positions  540 to  834
  -> Neither malonyl-CoA nor methylmalonyl-CoA specific

In sequence 'sp|Q9Y8A5|LOVB_ASPTE':
- Found PKSI-AT domain at positions  562 to  585
  -> Domain likely too short
- Found PKSI-AT domain at positions  651 to  854
  -> Neither malonyl-CoA nor methylmalonyl-CoA specific

In sequence 'sp|Q54FI3|STLB_DICDI':
- Found PKSI-AT domain at positions  625 to  726
  -> Domain likely too short
- Found PKSI-AT domain at positions  766 to  838
  -> Domain likely too short
- Found PKSI-AT domain at positions  880 to  944
  -> Domain likely too short
```

3.3 API Reference

3.3.1 HMMER

Reimplementation of HMMER binaries with the PyHMMER API.

pyhmmer.hmmer.**hmmsearch**(*queries*, *sequences*, *cpus=0*, *callback=None*, ***options*)

Search HMM profiles against a sequence database.

Parameters

- **queries** (iterable of *HMM*) – The query HMMs to search in the database.
- **sequences** (collection of *DigitalSequence*) – A database of sequences to query.
- **cpus** (int) – The number of threads to run in parallel. Pass `1` to run everything in the main thread, `0` to automatically select a suitable number (using `psutil.cpu_count`), or any positive number otherwise.
- **callback** (*callable*) – A callback that is called everytime a query is processed with two arguments: the query, and the total number of queries. This can be used to display progress in UI.

Yields *TopHits* – An object reporting *top hits* for each query, in the same order the queries were passed in the input.

Raises

- **AlphabetMismatch** – When any of the query HMMs
- **and the sequences do not share the same alphabet.** –

Note: Any additional arguments passed to the *hmmsearch* function will be passed transparently to the *Pipeline* to be created.

New in version 0.1.0.

pyhmmer.hmmer.**phmmer**(*queries: Iterable[pyhmmer.easel.DigitalMSA]*, *sequences: Collection[pyhmmer.easel.DigitalSequence]*, *cpus: int = 0*, *callback: Optional[Callable[[pyhmmer.easel.DigitalMSA, int], None]] = None*, *builder: Optional[pyhmmer.plan7.Builder] = None*, ***options: Any*) → Iterator[*pyhmmer.plan7.TopHits*]

Search protein sequences against a sequence database.

Parameters

- **queries** (iterable of `DigitalSequence` or `DigitalMSA`) – The query sequences to search in the database.
- **sequences** (collection of *`DigitalSequence`*) – A database of sequences to query.
- **cpus** (`int`) – The number of threads to run in parallel. Pass `1` to run everything in the main thread, `0` to automatically select a suitable number (using `psutil.cpu_count`), or any positive number otherwise.
- **callback** (*`callable`*) – A callback that is called everytime a query is processed with two arguments: the query, and the total number of queries. This can be used to display progress in UI.
- **builder** (*`Builder`*, optional) – A builder to configure how the queries are converted to HMMs. Passing `None` will create a default instance.

Yields *`TopHits`* – A *top hits* instance for each query, in the same order the queries were passed in the input.

Note: Any additional keyword arguments passed to the *`phmmer`* function will be passed transparently to the *`Pipeline`* to be created in each worker thread.

New in version 0.2.0.

Changed in version 0.3.0: Allow using `DigitalMSA` queries.

pyhmmer.hmmer.**nhmmer**(*queries: Iterable[pyhmmer.easel.DigitalMSA]*, *sequences: Collection[pyhmmer.easel.DigitalSequence]*, *cpus: int = 0*, *callback: Optional[Callable[[pyhmmer.easel.DigitalMSA, int], None]] = None*, *builder: Optional[pyhmmer.plan7.Builder] = None*, ***options: Any*) → Iterator[*pyhmmer.plan7.TopHits*]

Search protein sequences against a sequence database.

See also:

The equivalent function for proteins, *`phmmer`*.

New in version 0.3.0.

pyhmmer.hmmer.**hmmpress**(*hmms*, *output*)

Press several HMMs into a database.

Calling this function will create 4 files at the given location: `{output}.h3p` (containing the optimized profiles), `{output}.h3m` (containing the binary HMMs), `{output}.h3f` (containing the MSV parameters), and `{output}.h3i` (the SSI index mapping the previous files).

Parameters

- **hmms** (iterable of *`HMM`*) – The HMMs to be pressed together in the file.
- **output** (`str` or `os.PathLike`) – The path to an output location where to write the different files.

3.3.2 Easel

High-level interface to the Easel C library.

Easel is a library developed by the Eddy/Rivas Lab to facilitate the development of biological software in C. It is used by HMMER and Infernal.

Alphabet

class pyhmmer.easel.**Alphabet**

A biological alphabet, including additional marker symbols.

This type is used to share an alphabet to several objects in the `easel` and `plan7` modules. Reference counting helps sharing the same instance everywhere, instead of reallocating memory every time an alphabet is needed.

Use the factory class methods to obtain a default *Alphabet* for one of the three standard biological alphabets:

```
>>> dna = Alphabet.dna()
>>> rna = Alphabet.rna()
>>> aa  = Alphabet.amino()
```

amino()

Create a default amino-acid alphabet.

dna()

Create a default DNA alphabet.

rna()

Create a default RNA alphabet.

K

The alphabet size, counting only actual alphabet symbols.

Example

```
>>> Alphabet.dna().K
4
>>> Alphabet.amino().K
20
```

Type int

Kp

The complete alphabet size, including marker symbols.

Example

```
>>> Alphabet.dna().Kp
18
>>> Alphabet.amino().Kp
29
```

Type int

symbols
The symbols composing the alphabet.

Example

```
>>> Alphabet.dna().symbols
'ACGT-RYMKSWHBVDN*~'
>>> Alphabet.rna().symbols
'ACGU-RYMKSWHBVDN*~'
```

Type str

Bitfield

class pyhmmer.easel.**Bitfield**
A statically sized sequence of booleans stored as a packed bitfield.

A bitfield is instantiated with a fixed length, and all booleans are set to False by default:

```
>>> bitfield = Bitfield(8)
>>> len(bitfield)
8
>>> bitfield[0]
False
```

Use indexing to access and edit individual bits:

```
>>> bitfield[0] = True
>>> bitfield[0]
True
>>> bitfield[0] = False
>>> bitfield[0]
False
```

__init__(*length*)
Create a new bitfield with the given length.

count(*value=True*)
Count the number occurrences of value in the bitfield.

If no argument is given, counts the number of True occurences.

Example

```
>>> bitfield = Bitfield(8)
>>> bitfield.count(False)
8
>>> bitfield[0] = bitfield[1] = True
>>> bitfield.count()
2
```

toggle(*index*)
Switch the value of one single bit.

Example

```
>>> bitfield = Bitfield(8)
>>> bitfield[0]
False
>>> bitfield.toggle(0)
>>> bitfield[0]
True
>>> bitfield.toggle(0)
>>> bitfield[0]
False
```

KeyHash

class pyhmmer.easel.**KeyHash**

A dynamically resized container to store byte keys using a hash table.

Internally uses Bob Jenkins' *one at a time* hash, a simple and efficient hash function published in 1997 that exhibits avalanche behaviour.

Example

Add new keys to the key hash using the *add* method like you would with a Python `set`:

```
>>> kh = KeyHash()
>>> kh.add(b"key")
0
```

Check if a key hash contains a given key:

```
>>> b"key" in kh
True
>>> b"missing" in kh
False
```

Get the index associated with a key using the indexing notation:

```
>>> kh[b"key"]
0
>>> kh[b"missing"]
Traceback (most recent call last):
  ...
KeyError: b'missing'
```

See also:

The Wikipedia article for Bob Jenkins' hash functions: https://en.wikipedia.org/wiki/Jenkins_hash_function

__init__()

Create a new empty key-hash collection.

add(*item*)

Add a new key to the hash table, and return its index.

If `key` was already in the hash table, the previous index is returned:

```
>>> kh = KeyHash()
>>> kh.add(b"first")
0
>>> kh.add(b"second")
1
>>> kh.add(b"first")
0
```

Parameters **key** (bytes) – The key to add to the hash table.

Returns int – The index corresponding to the added key.

New in version 0.3.0.

clear()
Remove all entries from the collection.

copy()
Create and return an exact copy of this mapping.

Example

```
>>> kh = KeyHash()
>>> kh.add(b"key")
0
>>> copy = kh.copy()
>>> b"key" in copy
True
```

Multiple Sequence Alignment

class pyhmmer.easel.**MSA**
An abstract alignment of multiple sequences.

Hint: Use `len(msa)` to get the number of columns in the alignment, and `len(msa.sequences)` to get the number of sequences (i.e. the number of rows).

---

checksum()
Calculate a 32-bit checksum for the multiple sequence alignment.

write(*fh*, *format*)
Write the multiple sequence alignement to a file handle.

Parameters

- **fh** (io.IOBase) – A Python file handle, opened in binary mode.
- **format** (str) – The name of the multiple sequence alignment file format to use.

New in version 0.3.0.

accession
The accession of the alignment, if any.

Type bytes or None

author
The author of the alignment, if any.

Type bytes or None

description
The description of the sequence, if any.

Type bytes or None

name
The name of the alignment, if any.

Type bytes or None

class pyhmmer.easel.**TextMSA**(*MSA*)
A multiple sequence alignement stored in text mode.

__init__(*name=None*, *description=None*, *accession=None*, *sequences=None*, *author=None*)
Create a new text-mode alignment with the given sequences.

Parameters

- **name** (bytes, optional) – The name of the alignment, if any.
- **description** (bytes, optional) – The description of the alignment, if any.
- **accession** (bytes, optional) – The accession of the alignment, if any.
- **sequences** (iterable of *TextSequence*) – The sequences to store in the multiple sequence alignment. All sequences must have the same length. They also need to have distinct names.
- **author** (bytes, optional) – The author of the alignment, often used to record the aligner it was created with.

Raises

- **ValueError** – When the alignment cannot be created from the given sequences.
- **TypeError** – When sequences is not an iterable of *TextSequence* objects.

Example

```
>>> s1 = TextSequence(name=b"seq1", sequence="ATGC")
>>> s2 = TextSequence(name=b"seq2", sequence="ATGC")
>>> msa = TextMSA(name=b"msa", sequences=[s1, s2])
>>> len(msa)
4
```

Changed in version 0.3.0: Allow creating an alignment from an iterable of *TextSequence*.

copy()
Duplicate the text sequence alignment, and return the copy.

digitize(*alphabet*)
Convert the text alignment to a digital alignment using alphabet.

Returns *DigitalMSA* – An alignment in digital mode containing the same sequences digitized with alphabet.

sequences
A view of the sequences in the alignment.

This property lets you access the individual sequences in the multiple sequence alignment as *TextSequence* instances.

Example

Query the number of sequences in the alignment with `len`, or access individual members via indexing notation:

```
>>> s1 = TextSequence(name=b"seq1", sequence="ATGC")
>>> s2 = TextSequence(name=b"seq2", sequence="ATGC")
>>> msa = TextMSA(name=b"msa", sequences=[s1, s2])
>>> len(msa.sequences)
2
>>> msa.sequences[0].name
b'seq1'
```

Caution: Sequences in the list are copies, so editing their attributes will have no effect on the alignment:

```
>>> msa.sequences[0].name
b'seq1'
>>> msa.sequences[0].name = b"seq1bis"
>>> msa.sequences[0].name
b'seq1'
```

Support for this feature will be added in a future version, but can be circumvented for now by forcingly setting the updated version of the object:

```
>>> seq = msa.sequences[0]
>>> seq.name = b"seq1bis"
>>> msa.sequences[0] = seq
>>> msa.sequences[0].name
b'seq1bis'
```

New in version 0.3.0.

Type _TextMSASequences

class pyhmmer.easel.**DigitalMSA**(*MSA*)
A multiple sequence alignment stored in digital mode.

alphabet
The biological alphabet used to encode this sequence alignment to digits.

Type *Alphabet*

__init__(*alphabet*, *name=None*, *description=None*, *accession=None*, *sequences=None*, *author=None*)
Create a new digital-mode alignment with the given `sequences`.

Parameters

- **alphabet** (*Alphabet*) – The alphabet of the alignmed sequences.
- **name** (bytes, optional) – The name of the alignment, if any.

- **description** (bytes, optional) – The description of the alignment, if any.
- **accession** (bytes, optional) – The accession of the alignment, if any.
- **sequences** (iterable of *DigitalSequence*) – The sequences to store in the multiple sequence alignment. All sequences must have the same length and alphabet. They also need to have distinct names set.
- **author** (bytes, optional) – The author of the alignment, often used to record the aligner it was created with.

Changed in version 0.3.0: Allow creating an alignment from an iterable of *DigitalSequence*.

copy()
Duplicate the digital sequence alignment, and return the copy.

textize()
Convert the digital alignment to a text alignment.

Returns *TextMSA* – A copy of the alignment in text-mode.

New in version 0.3.0.

sequences
A view of the sequences in the alignment.

This property lets you access the individual sequences in the multiple sequence alignment as *DigitalSequence* instances.

See also:

The documentation for the *TextMSA.sequences* property, which contains some additional information.

New in version 0.3.0.

Type _DigitalMSASequences

Sequence

class pyhmmer.easel.**Sequence**
An abstract biological sequence with some associated metadata.

Easel provides two different mode to store a sequence: text, or digital. In the HMMER code, changing from one mode to another mode is done in place, which allows recycling memory. However, doing so can be confusing since there is no way to know statically the representation of a sequence.

To avoid this, pyhmmer provides two subclasses of the *Sequence* abstract class to maintain the mode contract: *TextSequence* and *DigitalSequence*. Functions expecting sequences in digital format, like pyhmmer.hmmsearch, can then use Python type system to make sure they receive sequences in the right mode. This allows type checkers such as mypy to detect potential contract breaches at compile-time.

checksum()
Calculate a 32-bit checksum for the sequence.

clear()
Reinitialize the sequence for re-use.

copy()
Duplicate the sequence, and return the copy.

write(*fh*)
Write the sequence alignement to a file handle, in FASTA format.

Parameters **fh** (io.IOBase) – A Python file handle, opened in binary mode.

New in version 0.3.0.

accession
The accession of the sequence.

Type bytes

description
The description of the sequence.

Type bytes

name
The name of the sequence.

Type bytes

source
The source of the sequence, if any.

Type bytes

class pyhmmer.easel.**TextSequence**(*Sequence*)
A biological sequence stored in text mode.

Hint: Use the sequence property to access the sequence letters as a Python string.

__init__(*name=None*, *description=None*, *accession=None*, *sequence=None*, *source=None*)
Create a new text-mode sequence with the given attributes.

copy()
Duplicate the text sequence, and return the copy.

digitize(*alphabet*)
Convert the text sequence to a digital sequence using alphabet.

Returns *DigitalSequence* – A copy of the sequence in digital-model, digitized with alphabet.

reverse_complement()
Build the reverse complement of the sequence.

This method assumes that the sequence alphabet is IUPAC/DNA. If the sequence contains any unknown letters, they will be replaced by N in the reverse-complement.

Parameters **inplace** (bool) – Whether or not to copy the sequence before computing its reverse complement. With False (the default), the method will return a copy of the sequence that has been reverse-complemented. With True, it will reverse-complement inplace and return None.

Raises **UserWarning** – When the sequence contains unknown characters.

Example

```
>>> seq = TextSequence(sequence="ATGC")
>>> seq.reverse_complement().sequence
'GCAT'
```

> **Caution:** The copy made when `inplace` is `False` is an exact copy, so the `name`, `description` and `accession` of the copy will be the same. This could lead to duplicates if you're not careful!

New in version 0.3.0.

sequence
The raw sequence letters, as a Python string.

Type `str`

class pyhmmer.easel.**DigitalSequence**(*Sequence*)
A biological sequence stored in digital mode.

alphabet
The biological alphabet used to encode this sequence to digits.

Type *Alphabet*, *readonly*

Hint: Use the `sequence` property to access the sequence digits as a memory view, allowing to access the individual bytes. This can be combined with `numpy.asarray` to get the sequence as an array with zero-copy.

__init__(*alphabet*, *name=None*, *description=None*, *accession=None*, *sequence=None*, *source=None*)
Create a new digital-mode sequence with the given attributes.

New in version 0.1.4.

copy()
Duplicate the digital sequence, and return the copy.

reverse_complement()
Build the reverse complement of the sequence.

Parameters **inplace** (`bool`) – Whether or not to copy the sequence before computing its reverse complement. With `False` (the default), the method will return a copy of the sequence that has been reverse-complemented. With `True`, it will reverse-complement inplace and return `None`.

Raises

- **ValueError** – When the alphabet of the *DigitalSequence* does
- **not have a complement mapping set (e.g., Alphabet.amino)** –

> **Caution:** The copy made when `inplace` is `False` is an exact copy, so the `name`, `description` and `accession` of the copy will be the same. This could lead to duplicates if you're not careful!

New in version 0.3.0.

textize()
Convert the digital sequence to a text sequence.

Returns *TextSequence* – A copy of the sequence in text-mode.

New in version 0.1.4.

sequence
The raw sequence digits, as a memory view.

Type memoryview

Sequence File

class pyhmmer.easel.**SequenceFile**

A wrapper around a sequence file, containing unaligned sequences.

This class supports reading sequences stored in different formats, such as FASTA, GenBank or EMBL. The format of each file can be automatically detected, but it is also possible to pass an explicit format specifier when the *SequenceFile* is instantiated.

New in version 0.2.0: The alphabet attribute.

__init__(*file*, *format=None*)

Create a new sequence file parser wrapping the given file.

Parameters

- **file** (str) – The path to a file containing sequences in one of the supported file formats.
- **format** (str, optional) – The format of the file, or None to autodetect. Supported values are: fasta, embl, genbank, ddbj, uniprot, ncbi, daemon, hmmpgmd, fmindex.

close()

Close the file and free the resources used by the parser.

guess_alphabet()

Guess the alphabet of an open *SequenceFile*.

This method tries to guess the alphabet of a sequence file by inspecting the first sequence in the file. It returns the alphabet, or None if the file alphabet cannot be reliably guessed.

Raises

- **EOFError** – if the file is empty.
- **OSError** – if a parse error occurred.
- **ValueError** – if this methods is called after the file was closed.

parse(*buffer*, *format*)

Parse a sequence from a binary buffer using the given format.

parseinto(*seq*, *buffer*, *format*)

Parse a sequence from a binary buffer into seq.

read(*skip_info=False*, *skip_sequence=False*)

Read the next sequence from the file.

Parameters

- **skip_info** (bool) – Pass True to disable reading the sequence *metadata*, and only read the sequence *letters*. Defaults to False.
- **skip_sequence** (bool) – Pass True to disable reading the sequence *letters*, and only read the sequence *metadata*. Defaults to False.

Returns *Sequence* – The next sequence in the file, or None if all sequences were read from the file.

Raises **ValueError** – When attempting to read a sequence from a closed file, or when the file could not be parsed.

Hint: This method allocates a new sequence, which is not efficient in case the sequences are being read within a tight loop. Use *SequenceFile.readinto* with an already initialized *Sequence* if you can to recycle the internal buffers.

readinto(*seq*, *skip_info=False*, *skip_sequence=False*)
Read the next sequence from the file, using `seq` to store data.

Parameters

- **seq** (*Sequence*) – A sequence object to use to store the next entry in the file. If this sequence was used before, it must be properly reset (using the *Sequence.clear* method) before using it again with *readinto*.
- **skip_info** (`bool`) – Pass `True` to disable reading the sequence *metadata*, and only read the sequence *letters*. Defaults to False`.
- **skip_sequence** (`bool`) – Pass `True` to disable reading the sequence *letters*, and only read the sequence *metadata*. Defaults to `False`.

Returns *Sequence* – A reference to `seq` that was passed as an input, or `None` if no sequences are left in the file.

Raises **ValueError** – When attempting to read a sequence from a closed file, or when the file could not be parsed.

Example

Use *SequenceFile.readinto* to loop over the sequences in a file while recycling the same *Sequence* buffer:

```
>>> with SequenceFile("vendor/hmmer/testsuite/ecori.fa") as sf:
...     seq = TextSequence()
...     while sf.readinto(seq) is not None:
...         # ... process seq here ... #
...         seq.clear()
```

set_digital(*alphabet*)
Set the *SequenceFile* to read in digital mode with `alphabet`.

This method can be called even after the first sequences have been read; it only affects subsequent sequences in the file.

Sequence / Subsequence Index

class pyhmmer.easel.**SSIReader**
A read-only handler for sequence/subsequence index file.

class Entry(*fd*, *record_offset*, *data_offset*, *record_length*)

property data_offset
Alias for field number 2

property fd
Alias for field number 0

property record_length
Alias for field number 3

property record_offset
Alias for field number 1

class FileInfo(*name*, *format*)

property format
Alias for field number 1

property name
Alias for field number 0

__init__(*file*)
Create a new SSI file reader for the file at the given location.

Parameters **file** (str) – The path to a sequence/subsequence index file to read.

close()
Close the SSI file reader.

file_info(*fd*)
Retrieve the *FileInfo* of the descriptor.

find_name(*key*)
Retrieve the *Entry* for the given name.

class pyhmmer.easel.**SSIWriter**
A writer for sequence/subsequence index files.

__init__(*file*)
Create a new SSI file write for the file at the given location.

Parameters

- **file** (str) – The path to a sequence/subsequence index file to write.
- **exclusive** (bool) – Whether or not to create a file if one does not exist.

Raises

- **FileNotFoundError** – When the path to the file cannot be resolved.
- **FileExistsError** – When the file exists and exclusive is True.

add_alias(*alias*, *key*)
Make alias an alias of key in the index.

add_file(*filename*, *format=0*)
Add a new file to the index.

Parameters

- **filename** (*str*) – The name of the file to register.
- **format** (*int*) – A format code to associate with the file, or *0*.

Returns int – The filehandle associated with the new indexed file.

add_key(*key*, *fd*, *record_offset*, *data_offset=0*, *record_length=0*)
Add a new entry to the index with the given key.

close()
Close the SSI file writer.

3.3.3 Plan7

High-level interface to the Plan7 data model.

Plan7 is the model architecture used by HMMER since HMMER2.

See also:

Details about the Plan 7 architecture in the HMMER documentation.

Alignment

class pyhmmer.plan7.**Alignment**

A single alignment of a sequence to a profile.

hmm_accession

The accession of the query, or its name if it has none.

New in version 0.1.4.

Type bytes

hmm_from

The start coordinate of the alignment in the query HMM.

Type int

hmm_name

The name of the query HMM.

Type bytes

hmm_sequence

The sequence of the query HMM in the alignment.

Type str

hmm_to

The end coordinate of the alignment in the query HMM.

Type int

identity_sequence

The identity sequence between the query and the target.

Type str

target_from

The start coordinate of the alignment in the target sequence.

Type int

target_name

The name of the target sequence.

Type bytes

target_sequence

The sequence of the target sequence in the alignment.

Type str

target_to

The end coordinate of the alignment in the target sequence.

Type int

Background Model

class pyhmmer.plan7.**Background**
The null background model of HMMER.

__init__(*alphabet*, *uniform=False*)
Create a new background model for the given `alphabet`.

Parameters

- **alphabet** (*pyhmmer.easel.Alphabet*) – The alphabet to create the background model with.
- **uniform** (bool) – Whether or not to create the null model with uniform frequencies. Defaults to `False`.

copy()
Create a copy of the null model with the same parameters.

L
The mean of the null model length distribution, in residues.

Type int

Builder

class pyhmmer.plan7.**Builder**
A factory for constructing new HMMs from raw sequences.

New in version 0.2.0.

__init__(*alphabet*, *, *architecture='fast'*, *weighting='pb'*, *effective_number='entropy'*, *prior_scheme='alpha'*, *symfrac=0.5*, *fragthresh=0.5*, *wid=0.62*, *esigma=45.0*, *eid=0.62*, *EmL=200*, *EmN=200*, *EvL=200*, *EvN=200*, *EfL=100*, *EfN=200*, *Eft=0.04*, *seed=42*, *ere=None*, *popen=None*, *pextend=None*)
Create a new sequence builder with the given configuration.

Parameters **alphabet** (*Alphabet*) – The alphabet the builder expects the sequences to be in.

Keyword Arguments

- **popen** (float) – The *gap open* probability to use with the score system. Default depends on the alphabet: *0.02* for proteins, *0.03125* for nucleotides.
- **pextend** (float) – The *gap extend* probability to use with the score system. Default depends on the alphabet: *0.4* for proteins, *0.75* for nucleotides.

build(*sequence*, *background*)
Build a new HMM from `sequence` using the builder configuration.

Parameters

- **sequence** (*DigitalSequence*) – A single biological sequence in digital mode to build a HMM with.
- **background** (pyhmmer.plan7.background) – The background model to use to create the HMM.

Raises **ValueError** – When either `sequence` or `background` have the wrong alphabet for this builder.

build_msa(*msa*, *background*)
Build a new HMM from `msa` using the builder configuration.

Parameters

- **msa** (*DigitalMSA*) – A multiple sequence alignment in digital mode to build a HMM with.
- **background** (`pyhmmer.plan7.background`) – The background model to use to create the HMM.

Raises **ValueError** – When either `msa` or `background` have the wrong alphabet for this builder.

New in version 0.3.0.

copy()
Create a duplicate *Builder* instance with the same arguments.

seed
The seed used by the internal random number generator.

Setting the seed will effectively reinitialize the internal RNG. In the special case the seed is *0*, a one-time arbitrary seed will be chosen and the RNG will no be reseeded for reproducibility.

Type `int`

Domains

class pyhmmer.plan7.**Domain**
A single domain in a query *Hit*.

c_evalue
The conditional e-value for the domain.

Type `float`

i_evalue
The independent e-value for the domain.

Type `float`

score
The overall score in bits, null-corrected.

Type `float`

class pyhmmer.plan7.**Domains**
A read-only view over the domains of a single *Hit*.

Hits

class pyhmmer.plan7.**Hit**
A high-scoring database hit found by the comparison pipeline.

accession
The accession of the database hit, if any.

Type bytes or None

description
The description of the database hit, if any.

Type bytes or None

domains
The list of domains aligned to this hit.

Type *Domains*

evalue
The e-value of the hit.

Type float

name
The name of the database hit.

Type bytes

pre_score
Bit score of the sequence before *null2* correction.

Type float

score
Bit score of the sequence with all domains after correction.

Type float

class pyhmmer.plan7.**TopHits**
A ranked list of top-scoring hits.

TopHits are thresholded using the parameters from the pipeline, and are sorted by key when you obtain them from a *Pipeline* instance:

```
>>> abc = thioesterase.alphabet
>>> hits = Pipeline(abc).search_hmm(thioesterase, proteins)
>>> hits.is_sorted()
True
```

Use len to query the number of top hits, and the usual indexing notation to extract a particular *Hit*:

```
>>> len(hits)
1
>>> hits[0].name
b'938293.PRJEB85.HG003687_113'
```

__init__()
Create an empty *TopHits* instance.

clear()
Free internals to allow reusing for a new pipeline run.

is_sorted(*by='key'*)
Check whether or not the hits are sorted with the given method.

See *sort* for a list of allowed values for the `by` argument.

sort(*by='key'*)
Sort hits in the current instance using the given method.

Parameters **by** (`str`) – The comparison method to use to compare hits. Allowed values are: `key` (the default) to sort by key, or `seqidx` to sort by sequence index and alignment position.

to_msa(*alphabet*, *trim=False*, *digitize=False*, *all_consensus_cols=False*)
Create multiple alignment of all included domains.

Parameters

- **alphabet** (*Alphabet*) – The alphabet of the HMM this *TopHits* was obtained from. It is required to convert back hits to single sequences.
- **trim** (`bool`) – Trim off any residues that get assigned to flanking N and C states (in profile traces) or I_0 and I_m (in core traces).
- **digitize** (`bool`) – If set to `True`, returns a `DigitalMSA` instead of a `TextMSA`.
- **all_consensus_cols** (`bool`) – Force a column to be created for every consensus column in the model, even if it means having all gap character in a column.

Returns *MSA* – A multiple sequence alignment containing the reported hits, either a `TextMSA` or a `DigitalMSA` depending on the value of the `digitize` argument.

New in version 0.3.0.

included
The number of hits that are above the inclusion threshold.

Type `int`

reported
The number of hits that are above the reporting threshold.

Type `int`

HMM

class pyhmmer.plan7.**HMM**
A data structure storing the Plan7 Hidden Markov Model.

__init__(*M*, *alphabet*)
Create a new HMM from scratch.

Parameters

- **M** (`int`) – The length of the model (i.e. the number of nodes).
- **alphabet** (*Alphabet*) – The alphabet of the model.

copy()
Return a copy of the HMM with the exact same configuration.

New in version 0.3.0.

write(*fh*, *binary=False*)
Write the HMM to a file handle.

Parameters

- **fh** (io.IOBase) – A Python file handle, opened in binary mode (this must be the case even with binary=False, since the C code will emit bytes in either case).
- **binary** (bool) – Pass False to emit the file in ASCII mode using the latest supported HMMER format, or True to use the binary HMMER3 format.

zero()
Set all parameters to zero, including model composition.

M
The length of the model (i.e. the number of nodes).

Type int

accession
The accession of the HMM, if any.

Type bytes or None

checksum
The 32-bit checksum of the HMM, if any.

The checksum if calculated from the alignment the HMM was created from, and was introduced in more recent HMM formats. This means some *HMM* objects may have a non-None checksum.

New in version 0.2.1.

Changed in version 0.3.1: Returns None if the HMM flag for the checksum is not set.

Type int or None

command_line
The command line that built the model.

For HMMs created with *Builder*, this defaults to sys.argv. It can however be set to any string, including multiline to show successive commands.

Example

```
>>> print(thioesterase.command_line)
hmmbuild Thioesterase.hmm Thioesterase.fa
hmmcalibrate Thioesterase.hmm
```

New in version 0.3.1.

Type str or None

consensus
The consensus residue line of the HMM, if set.

New in version 0.3.0.

Type str or None

consensus_accessibility
The consensus accessibility of the HMM, if any.

New in version 0.3.1.

Type str or None

consensus_structure
The consensus structure of the HMM, if any.

New in version 0.3.1.

Type `str` or `None`

description
The description of the HMM, if any.

Type `bytes` or `None`

insert_emissions
The insert emissions of the model.

The returned memoryview exposes a matrix of dimensions (M, K), with one row per node and one column per alphabet symbol.

Hint: Use `numpy.asarray` to convert the memoryview to a 2D aray:

```
>>> i = thioesterase.insert_emissions
>>> numpy.asarray(i).reshape((thioesterase.M, thioesterase.alphabet.K))
array([[...]], dtype=float32)
```

New in version 0.3.1.

Type `memoryview` of `float`

match_emissions
The match emissions of the model.

The returned memory view exposes a matrix of dimensions (M, K), with one row per node, and one column per alphabet symbol.

Hint: Use `numpy.asarray` to convert the memory view to a 2D array:

```
>>> m = thioesterase.match_emissions
>>> numpy.asarray(m).reshape((thioesterase.M, thioesterase.alphabet.K))
array([[...]], dtype=float32)
```

New in version 0.3.1.

Type `memoryview` of `float`

model_mask
The model mask line from the alignment, if any.

New in version 0.3.1.

Type `str` or `None`

name
The name of the HMM, if any.

Type `bytes` or `None`

nseq
The number of training sequences used, if any.

If the HMM was created from a multiple sequence alignment, this corresponds to the number of sequences in the MSA.

Example

```
>>> thioesterase.nseq
278
```

New in version 0.3.1.

Type int or None

nseq_effective
The number of effective sequences used, if any.

New in version 0.3.1.

Type float or None

reference
The reference line from the alignment, if any.

This is relevant if the HMM was built from a multiple sequence alignment (e.g. by *Builder.build_msa*, or by an external hmmbuild pipeline run).

New in version 0.3.1.

Type str or None

transition_probabilities
The transition probabilities of the model.

The returned memory view exposes a matrix of dimensions $(M + 1, 7)$, with one row per node (plus one extra row for the entry probabilities), and one column per transition.

Hint: Use numpy.asarray to convert the memory view to a 2D array:

```
>>> t = thioesterase.transition_probabilities
>>> numpy.asarray(t).reshape((thioesterase.M+1, 7))
array([[...]], dtype=float32)
```

New in version 0.3.1.

Type memoryview of float

HMM File

class pyhmmer.plan7.**HMMFile**
A wrapper around a file (or database), storing serialized HMMs.

__init__(*file*, *db=True*)
Create a new HMM reader from the given file.

Parameters

- **file** (str or file-like object) – Either the path to a file containing the HMMs to read, or a file-like object opened in binary-mode.

- **db** (*bool*) – Set to `False` to force the parser to ignore the pressed HMM database if it finds one. Defaults to `False`.

close()
Close the HMM file and free resources.

This method has no effect if the file is already closed. It is called automatically if the *HMMFile* was used in a context:

```
>>> with HMMFile("tests/data/hmms/bin/PKSI-AT.h3m") as hmm_file:
...     hmm = next(hmm_file)
```

Pipeline

class pyhmmer.plan7.**Pipeline**
An HMMER3 accelerated sequence/profile comparison pipeline.

__init__(*alphabet*, *background=None*, ***, *bias_filter=True*, *report_e=10.0*, *null2=True*, *seed=42*, *Z=None*, *domZ=None*)
Instantiate and configure a new accelerated comparison pipeline.

Parameters

- **alphabet** (*Alphabet*) – The biological alphabet the of the HMMs and sequences that are going to be compared. Used to build the background model.
- **background** (*Background*, optional) – The background model to use with the pipeline, or `None` to create and use a default one. *The pipeline needs ownership of the background model, so any background model passed there will be copied.*

Keyword Arguments

- **bias_filter** (*bool*) – Whether or not to enable composition bias filter. Defaults to `True`.
- **null2** (*bool*) – Whether or not to compute biased composition score corrections. Defaults to `True`.
- **report_e** (*float*) – Report hits with e-value lower than or equal to this threshold in output. Defaults to `10.0`.
- **seed** (*int*, optional) – The seed to use with the random number generator. Pass *0* to use a one-time arbitrary seed, or `None` to keep the default seed from HMMER.

clear()
Reset the pipeline configuration to its default state.

scan_seq(*query*, *hmms*)
Run the pipeline using a query sequence against a profile database.

Parameters

- **query** (*DigitalSequence*) – The sequence object to use to query the profile database.
- **hmms** (iterable of *DigitalSequence*) – The HMM profiles to query. Pass a `HMMFile` instance to read from disk iteratively.

Returns *TopHits* – the hits found in the profile database.

Raises **AlphabetMismatch** – When the alphabet of the current pipeline does not match the alphabet of the given query or profile.

> **Caution:** In the current version, this method is not optimized to use the *pressed* database, even if it exists. This will cause the MSV and SSV filters to be rebuilt at each iteration, which could be slow. Consider at least pre-fetching the HMM database if calling this method several times in a row.

New in version v0.4.0.

search_hmm(*query*, *sequences*)
Run the pipeline using a query HMM against a sequence database.

Parameters

- **query** (*HMM*) – The HMM object to use to query the sequence database.
- **sequences** (iterable of *DigitalSequence*) – The sequences to query with the HMM. For instance, pass a *SequenceFile* in digital mode to read from disk iteratively.

Returns *TopHits* – the hits found in the sequence database.

Raises **AlphabetMismatch** – When the alphabet of the current pipeline does not match the alphabet of the given HMM.

New in version 0.2.0.

search_msa(*query*, *sequences*, *builder=None*)
Run the pipeline using a query alignment against a sequence database.

Parameters

- **query** (*DigitalMSA*) – The multiple sequence alignment to use to query the sequence database.
- **sequences** (iterable of *DigitalSequence*) – The sequences to query. Pass a `SequencesFile` instance in digital mode to read from disk iteratively.
- **builder** (*Builder*, optional) – A HMM builder to use to convert the query to a *HMM*. If `None` is given, it will use a default one.

Returns *TopHits* – the hits found in the sequence database.

Raises **AlphabetMismatch** – When the alphabet of the current pipeline does not match the alphabet of the given query.

New in version 0.3.0.

search_seq(*query*, *sequences*, *builder=None*)
Run the pipeline using a query sequence against a sequence database.

Parameters

- **query** (*DigitalSequence*) – The sequence object to use to query the sequence database.
- **sequences** (iterable of *DigitalSequence*) – The sequences to query. Pass a *SequenceFile* instance in digital mode to read from disk iteratively.
- **builder** (*Builder*, optional) – A HMM builder to use to convert the query to a *HMM*. If `None` is given, it will use a default one.

Returns *TopHits* – the hits found in the sequence database.

Raises **AlphabetMismatch** – When the alphabet of the current pipeline does not match the alphabet of the given query.

New in version 0.2.0.

Z
The number of effective targets searched.

It is used to compute the independent e-value for each domain, and for an entire hit. If `None`, the parameter number will be set automatically after all the comparisons have been done. Otherwise, it can be set to an arbitrary number.

Type `float` or `None`

domZ
The number of significant targets.

It is used to compute the conditional e-value for each domain. If `None`, the parameter number will be set automatically after all the comparisons have been done, and all hits have been thresholded. Otherwise, it can be set to an arbitrary number.

Type `float` or `None`

seed
The seed used by the internal random number generator.

New in version 0.2.0.

Type `int`

Profile

class pyhmmer.plan7.**Profile**
A Plan7 search profile.

__init__(*M*, *alphabet*)
Create a new profile for the given `alphabet`.

Parameters

- **M** (`int`) – The length of the profile, i.e. the number of nodes.
- **alphabet** (*Alphabet*) – The alphabet to use with this profile.

clear()
Clear internal buffers to reuse the profile without reallocation.

configure(*hmm*, *background*, *L*, *multihit=True*, *local=True*)
Configure a search profile using the given models.

Parameters

- **hmm** (*pyhmmer.plan7.HMM*) – The model HMM with core probabilities.
- **bg** (*pyhmmer.plan7.Background*) – The null background model.
- **L** (`int`) – The expected target sequence length.
- **multihit** (`bool`) – Whether or not to use multihit modes.
- **local** (`bool`) – Whether or not to use non-local modes.

copy()
Return a copy of the profile with the exact same configuration.

is_local()
Return whether or not the profile is in a local alignment mode.

is_multihit()
Returns whether or not the profile is in a multihit alignment mode.

optimized()
Convert the profile to a platform-specific optimized profile.

Returns *OptimizedProfile* – The platform-specific optimized profile built using the configuration of this profile.

L
The current configured target sequence length.

Type int

M
The length of the profile (i.e. the number of nodes).

New in version 0.3.0.

Type int

accession
The accession of the profile, if any.

New in version 0.3.0.

Type bytes or None

description
The description of the profile, if any.

New in version 0.3.0.

Type bytes or None

name
The name of the profile, if any.

New in version 0.3.0.

Type bytes or None

class pyhmmer.plan7.**OptimizedProfile**
An optimized profile that uses platform-specific instructions.

__init__(*M*, *alphabet*)
Create a new optimized profile from scratch.

Optimized profiles use platform-specific code to accelerate the various algorithms. Although you can allocate an optimized profile yourself, the only way to obtain a fully configured profile is to create it with the *Profile.optimized* method, after having configured the profile for a given HMM with *Profile.configure*.

Parameters

- **M** (int) – The length of the model (i.e. the number of nodes).
- **alphabet** (*Alphabet*) – The alphabet of the model.

copy()
Create an exact copy of the optimized profile.

is_local()
Return whether or not the profile is in a local alignment mode.

write(*fh_filter*, *fh_profile*)
Write an optimized profile to two separate files.

HMMER implements an acceleration pipeline using several scoring algorithms. Parameters for MSV (the *Multi ungapped Segment Viterbi*) are saved independently to the `fh_filter` handle, while the rest of the profile is saved to `fh_profile`.

3.3.4 Errors

Common errors and status codes for the `easel` and `hmmer` modules.

exception pyhmmer.errors.**AllocationError**(*MemoryError*)
A memory error that is caused by an unsuccessful allocation.

exception pyhmmer.errors.**UnexpectedError**(*RuntimeError*)
An unexpected error that happened in the C code.

As a user of this library, you should never see this exception being raised. If you do, please open an issue with steps to reproduce on the bug tracker, so that proper error handling can be added to the relevant part of the bindings.

exception pyhmmer.errors.**EaselError**(*RuntimeError*)
An error that was raised from the Easel code.

Cython bindings and Python interface to HMMER3.

HMMER is a biological sequence analysis tool that uses profile hidden Markov models to search for sequence homologs. HMMER3 is maintained by members of the the Eddy/Rivas Laboratory at Harvard University.

`pyhmmer` is a module, implemented using the Cython language, that provides bindings to HMMER3. It directly interacts with the HMMER internals, which has several advantages over CLI wrappers like hmmer-py.

3.4 Contributing to pyHMMER

For bug fixes or new features, please file an issue before submitting a pull request. If the change isn't trivial, it may be best to wait for feedback.

3.4.1 Setting up a local repository

Make sure you clone the repository in recursive mode, so you also get the wrapped code of Easel and HMMER, which are exposed as `git` submodules:

```
$ git clone --recursive https://github.com/althonos/pyhmmer
```

3.4.2 Running tests

Tests are written as usual Python unit tests with the `unittest` module of the standard library. Running them requires the extension to be built locally:

```
$ python setup.py build_ext --debug --inplace
$ python -m unittest discover -vv
```

3.4.3 Coding guidelines

This project targets Python 3.6 or later.

Python objects should be typed; since it is not supported by Cython, you must manually declare types in type stubs (`.pyi` files). In Python files, you can add type annotations to function signatures (supported in Python 3.5) or in variable assignments (supported from Python 3.6 onward).

Interfacing with C

When interfacing with C, and in particular with pointers, use assertions everywhere you assume the pointer to be non-NULL.

3.5 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

3.5.1 Unreleased

3.5.2 v0.3.1 - 2021-05-08

Added

- `Pipeline.scan_seq` method to query a database of profiles with one or more sequences.
- `transition_probabilities`, `match_emissions`, `insert_emissions` properties to the `HMM` class, providing access to the numerical parameters of the HMM.
- `consensus_structure` and `consensus_accessibility` properties to the `HMM` class to get consensus lines from the source alignment if the HMM was created from a MSA.
- `nseq` and `nseq_effective` properties to the `HMM` class to get the number of training sequences and effective sequences used to build the HMM.

Changed

- `HMM.checksum` is now `None` if the `p7H_CHKSUM` flag is not set.
- `Builder` methods will now record `sys.argv` when creating a HMM.

Fixed

- `HMM.write(..., binary=False)` crashing on HMMs without a consensus line. (#5). Fixed upstream in (EddyRivasLab/HMMER#236).
- `Pipeline.reset` mishandling the `Z` and `domZ` values if those were detected from the number of targets.
- `pyhmmer.hmmer` functions will not block until all results have been collected anymore when run in multi-threaded mode.

3.5.3 v0.3.0 - 2021-03-11

Added

- `easel.MSAFile` to read from a file containing
- `accession`, `author`, `name` and `description` properties to `easel.MSA` objects.
- `plan7.Builder.build_msa` to build a pHMM from a sequence alignment.
- Additional methods to `easel.KeyHash`, allowing to use it as a `dict`/`set` hybrid.
- `Sequence.write` and `MSA.write` methods to format a sequence or an alignment to a file handle.
- `plan7.TopHist.to_msa` method to convert all the top hits of a query against a database into a multiple sequence alignment.
- `easel.MSA.sequences` attribute to access individual sequences of an alignment using the `collections.abc.Sequence` interface.
- `easel.DigitalMSA.textize` method to convert a multiple sequence alignment in digital mode to its text-mode counterpart.
- Read-only `name`, `accession` and `description` properties to `plan7.Profile` showing attributes inherited from the HMM it was configured with.
- `plan7.HMM.consensus` property, allowing to access the consensus sequence of a pHMM.
- `plan7.HMM` equality implementation, using zero tolerance.
- `plan7.Pipeline.search_msa` to query a MSA against a sequence database.
- `easel.Sequence.reverse_complement` method allowing to reverse-complement inplace or to build a copy.
- `errors.AlphabetMismatch` exception for use in cases where an alphabet is expected but not matched by the input.
- `hmmer.nhmmer` function with the same behaviour as `hmmer.phmmer`, except it expects inputs with a DNA alphabet.

Fixed

- `plan7.Builder.copy` not copying some parameters correctly, causing `pyhmmer.hmmer.phmmer` to give inconsistent results in multithreaded mode.
- `easel.Bitfield` not properly handling index overflows.
- Documentation not rendering for the `__init__` method of all classes.

Changed

- `plan7.Builder` gap-open and gap-extend probabilities are now set on instantiation and depend on the alphabet type.
- Constructors for `easel.TextMSA` and `easel.DigitalMSA`, which can now be given an iterable of `easel.Sequence` objects to store in the alignment.

Removed

- Unimplemented `easel.SequenceFile.fetch` and `easel.SequenceFile.fetchinto` methods.

3.5.4 v0.2.2 - 2021-03-04

Fixed

- Linking issues on OSX caused by aggressive stripping of intermediate libraries.
- `plan7.Builder` RNG not reseeding between different HMMs.

3.5.5 v0.2.1 - 2021-01-29

Added

- `pyhmmer.plan7.HMM.checksum` property to get the 32-bit checksum of an HMM.

3.5.6 v0.2.0 - 2021-01-21

Added

- `pyhmmer.plan7.Builder` class to handle building a HMM from a sequence.
- `Pipeline.search_seq` to query a sequence against a sequence database.
- `psutil` dependency to detect the most efficient thread count for `hmmsearch` based on the number of *physical* CPUs.
- `pyhmmer.hmmer.phmmer` function to run a search of query sequences against a sequence database.

Changed

- `Pipeline.search` was renamed to `Pipeline.search_hmm` for disambiguation.
- `libeasel.random` sequences do not require the GIL anymore.
- Public API now have proper signature annotations.

Fixed

- Inaccurate exception messages in `Pipeline.search_hmm`.
- Unneeded RNG reallocation, replaced with re-initialisation where possible.
- `SequenceFile.__next__` not working after being set in digital mode.
- `sequences` argument of `hmmsearch` now only requires a `typing.Collection[DigitalSequence]` instead of a `typing.Collection[Sequence]` (not more `__getitem__` needed).

Removed

- `hits` argument to `Pipeline.search_hmm` to reduce risk of issues with `TopHits` reuse.
- Broken alignment coordinates on `Domain` classes.

3.5.7 v0.1.4 - 2021-01-15

Added

- `DigitalSequence.textize` to convert a digital sequence to a text sequence.
- `DigitalSequence.__init__` method allowing to create a digital sequence from any object implementing the buffer protocol.
- `Alignment.hmm_accession` property to retrieve the accession of the HMM in an alignment.

3.5.8 v0.1.3 - 2021-01-08

Fixed

- Compilation issues in OSX-specific Cython code.

3.5.9 v0.1.2 - 2021-01-07

Fixed

- Required Cython files not being included in source distribution.

3.5.10 v0.1.1 - 2020-12-02

Fixed

- `HMMFile` calling `file.peek` without arguments, causing it to crash when passed some types, e.g. `gzip.GzipFile`.
- `HMMFile` failing to work with PyPy file objects because of a bug with their implementation of `readinto`.
- C/Python file object implementation using `strcpy` instead of `memcpy`, causing issues when null bytes were read.

3.5.11 v0.1.0 - 2020-12-01

Initial beta release.

Fixed

- `TextSequence` uses the sequence argument it's given on instantiation.
- Segmentation fault in `Sequence.__eq__` caused by implicit type conversion.
- Segmentation fault on `SequenceFile.read` failure.
- Missing type annotations for the `pyhmmer.easel` module.

3.5.12 v0.1.0-a5 - 2020-11-28

Added

- `Sequence.__len__` magic method so that `len(seq)` returns the number of letters in `seq`.
- Python file-handle support when opening an `pyhmmer.plan7.HMMFile`.
- Context manager protocol to `pyhmmer.easel.SSIWriter`.
- Type annotations for `pyhmmer.easel.SSIWriter`.
- `add_alias` to `pyhmmer.easel.SSIWriter`.
- `write` method to `pyhmmer.plan7.OptimizedProfile` to write an optimized profile in binary format.
- `offsets` property to interact with the disk offsets of a `pyhmmer.plan7.OptimizedProfile` instance.
- `pyhmmer.hmmer.hmmpress` emulating the `hmmpress` binary from HMMER.
- `M` property to `pyhmmer.plan7.HMM` exposing the number of nodes in the model.

Changed

- Bumped vendored Easel to `v0.48`.
- Bumped vendored HMMER to `v3.3.2`.
- `pyhmmer.plan7.HMMFile` will raise an `EOFError` when given an empty file.
- Renamed `length` property to `L` in `pyhmmer.plan7.Background`.

Fixed

- Segmentation fault when `close` method of `pyhmmer.easel.SSIWriter` was called more than once.
- `close` method of `pyhmmer.easel.SSIWriter` not writing the index contents.

3.5.13 v0.1.0-a4 - 2020-11-24

Added

- `MSA`, `TextMSA` and `DigitalMSA` classes representing a multiple sequence alignment to `pyhmmer.easel`.
- Methods and protocol to copy a `Sequence` and a `MSA`.
- `pyhmmer.plan7.OptimizedProfile` wrapping a platform-specific optimized profile.
- `SSIReader` and `SSIWriter` classes interacting with sequence/subsequence indices to `pyhmmer.easel`.

- Exception handler using Python exceptions to report Easel errors.

Changed

- `pyhmmer.hmmsearch` returns an iterator of `TopHits`, with one instance per `HMM` in the input.
- `pyhmmer.hmmsearch` properly raises errors happenning in the background threads without deadlock.
- `pyhmmer.plan7.Pipeline` recycles memory between `Pipeline.search` calls.

Fixed

- Missing type annotations for the `pyhmmer.errors` module.

Removed

- Unneeded or private methods from `pyhmmer.plan7`.

3.5.14 v0.1.0-a3 - 2020-11-19

Added

- `TextSequence` and `DigitalSequence` representing a `Sequence` in a given mode.
- E-value properties to `Hit` and `Domain`.
- `TopHits` now stores a reference to the pipeline it was obtained from.
- `Pipeline.Z` and `Pipeline.domZ` properties.
- Experimental pickling support to `Alphabet`.
- Experimental freelist to `Sequence` class to avoid allocation bottlenecks when iterating on a `SequenceFile` without recycling sequence buffers.

Changed

- Made `Sequence` an abstract base class.
- Additional `Pipeline` parameters can be passed as keyword arguments to `pyhmmer.hmmsearch`.
- `SequenceFile.read` can now be configured to skip reading the metadata or the content of a sequence.

Removed

- Redundant `SequenceFile` methods.

Fixed

- `doctest` loader crashing on Python 3.5.
- `TopHits.threshold` segfaulting when being called without prior `Tophits.sort` call
- Unknown `format` argument to `SequenceFile` constructor not raising the right error.

3.5.15 v0.1.0-a2 - 2020-11-12

Added

- Support for compilation on PowerPC big-endian platforms.
- Type annotations and stub files for Cython modules.

Changed

- `distutils` is now used to compile the package, instead of calling `autotools` and letting HMMER configure itself.
- `Bitfield.count` now allows passing an argument (for compatibility with `collections.abc.Sequence`).

3.5.16 v0.1.0-a1 - 2020-11-10

Initial alpha release (test deployment to PyPI).

CHAPTER

FOUR

RELATED PROJECT

If despite of all the advantages listed earlier, you would rather use HMMER through its CLI, this package will not be of great help. You should then check the hmmer-py package developed by Danilo Horta at the EMBL-EBI.

CHAPTER

FIVE

LICENSE

This library is provided under the MIT License. The HMMER3 and Easel code is available under the BSD 3-clause license, which allows redistribution of their sources in the `pyhmmer` distribution.

This project is in no way not affiliated, sponsored, or otherwise endorsed by the original HMMER authors. It was developed by Martin Larralde during his PhD project at the European Molecular Biology Laboratory in the Zeller team.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

Z