decaylanguage

Release 0.2.0

Henry Fredrick Schreiner III

Apr 24, 2024

CONTENTS

CHAPTER

ONE

INSTALLATION

At the command line:

```
pip install decaylanguage
```

Or:

```
pip install https://github.com/scikit-hep/decaylanguage.git
```

CHAPTER

TWO

USAGE

Particles are a key component when dealing with decays. Refer to the `Particle package`_ for how to deal with particles and PDG identification codes.

2.1 DecFiles

DecFiles describe decays. You can read files and query information about them.

2.2 DecayLanguage

The primary way to use `decaylanguage.decay` is through the module provided to read in a language file and produce an output. For example, for GooFit, call:

```
python -m decaylanguage.goofit models/DtoKpipipi_v2.txt
```

You can pipe the output to a file.

Examples of interaction with the API directly are provided in the `/notebooks` folder, including svg diagrams of lines.

CHAPTER

THREE

REFERENCE

3.1 decaylanguage

class decaylanguage.**DaughtersDict**(*iterable: dict[str, int] | list[str] | tuple[str] | str | None = None*, ***kwds: Any*)

Class holding a decay final state as a dictionary. It is a building block for the digital representation of full decay chains.

Note: This class assumes EvtGen particle names, though this assumption is only relevant for the `charge_conjugate` method. Otherwise, all other class methods smoothly deal with any kind of particle names (basically an iterable of strings).

Example

A final state such as 'K+ K- K- pi+ pi0' is stored as `{'K+': 1, 'K-': 2, 'pi+': 1, 'pi0': 1}`.

charge_conjugate(*pdg_name: bool = False*) → Self

Return the charge-conjugate final state.

Parameters

pdg_name (*str, optional, default=False*) – Input particle name is the PDG name, not the (default) EvtGen name.

Examples

```
>>> dd = DaughtersDict({'K+': 2, 'pi0': 1})
>>> dd.charge_conjugate()
<DaughtersDict: ['K-', 'K-', 'pi0']>
>>>
>>> dd = DaughtersDict({'K_S0': 1, 'pi0': 1})
>>> dd.charge_conjugate()
<DaughtersDict: ['K_S0', 'pi0']>
>>>
>>> dd = DaughtersDict({'K(S)0': 1, 'pi+': 1})  # PDG names!
>>> # 'K(S)0' unrecognised in charge conjugation unless specified that these are␣
↪PDG names
>>> dd.charge_conjugate()
```

(continues on next page)

(continued from previous page)

```
<DaughtersDict: ['ChargeConj(K(S)0)', 'pi-']>
>>> dd.charge_conjugate(pdg_name=True)
<DaughtersDict: ['K(S)0', 'pi-']>
```

classmethod fromkeys(*iterable*, *v=None*)

Create a new dictionary with keys from iterable and values set to value.

to_list() → list[str]

Return the daughters as an ordered list of names.

to_string() → str

Return the daughters as a string representation (ordered list of names).

class decaylanguage.**DecFileParser**(**filenames: str | PathLike[str]*)

The class to parse a .dec decay file.

Example

```
>>> dfp = DecFileParser('my-decay-file.dec')
>>> dfp.parse()
```

build_decay_chains(*mother: str*, *stable_particles: list[str] | set[str] | tuple[str] | tuple[()] = ()*) → dict[str, list[DecayModeDict]]

Iteratively build the entire decay chains of a given mother particle, optionally considering, on the fly, certain particles as stable. This way, for example, only the B -> D E F part in a decay chain A -> B (-> D E F (-> G H)) C can be trivially selected for inspection.

Parameters

- **mother** (*str*) – Input mother particle name.
- **stable_particles** (*iterable, optional, default=()*) – If provided, stops the decay-chain parsing, taking the "list" as particles to be considered stable.

Returns

out (*dict*) – Decay chain as a dictionary of the form {mother: [{'bf': float, 'fs': list, 'model': str, 'model_params': str}]} where 'bf' stands for the decay mode branching fraction, 'fs' is a list of final-state particle names (strings) and/or dictionaries of the same form as the decay chain above, 'model' is the model name, if found, else '', 'model_params' are the model parameters, if specified, else ''

Examples

```
>>> parser = DecFileParser('a-Dplus-decay-file.dec')
>>> parser.parse()
>>> parser.build_decay_chains('D+')
{'D+': [{'bf': 1.0,
   'fs': ['K-',
    'pi+',
    'pi+',
    {'pi0': [{'bf': 0.988228297,
       'fs': ['gamma', 'gamma'],
```

(continues on next page)

(continued from previous page)

```
       'model': 'PHSP',
       'model_params': ''},
      {'bf': 0.011738247,
       'fs': ['e+', 'e-', 'gamma'],
       'model': 'PI0_DALITZ',
       'model_params': ''},
      {'bf': 3.3392e-05,
      'fs': ['e+', 'e+', 'e-', 'e-'],
      'model': 'PHSP',
      'model_params': ''},
      {'bf': 6.5e-08, 'fs': ['e+', 'e-'], 'model': 'PHSP', 'model_params': ''}]}
↪],
   'model': 'PHSP',
   'model_params': ''}]}
>>> p.build_decay_chains('D+', stable_particles=['pi0'])
{'D+': [{'bf': 1.0, 'fs': ['K-', 'pi+', 'pi+', 'pi0'], 'model': 'PHSP', 'model_
↪params': ''}]}
```

dict_aliases() → dict[str, str]

Return a dictionary of all alias definitions in the input parsed file, of the form "Alias <NAME> <ALIAS>", as {'NAME1': ALIAS1, 'NAME2': ALIAS2, ...}.

dict_charge_conjugates() → dict[str, str]

Return a dictionary of all charge conjugate definitions in the input parsed file, of the form "ChargeConj <PARTICLE> <CC_PARTICLE>", as {'PARTICLE1': CC_PARTICLE1, 'PARTICLE2': CC_PARTICLE2, ...}.

dict_decays2copy() → dict[str, str]

Return a dictionary of all statements in the input parsed file defining a decay to be copied, of the form "CopyDecay <NAME> <DECAY_TO_COPY>", as {'NAME1': DECAY_TO_COPY1, 'NAME2': DECAY_TO_COPY2, ...}.

dict_definitions() → dict[str, float]

Return a dictionary of all definitions in the input parsed file, of the form "Define <NAME> <VALUE>", as {'NAME1': VALUE1, 'NAME2': VALUE2, ...}.

dict_jetset_definitions() → dict[str, dict[int, int | float | str]]

Return a dictionary of all JETSET definitions in the input parsed file, of the form "JetSetPar <MODULE>(<PARAMETER>)=<VALUE>" as {'MODULE1': {PARAMETER1: VALUE1, PARAMETER2: VALUE2, ...},

'MODULE2': {...}, ...}.

dict_lineshape_settings() → dict[str, dict[str, str | float]]

Return a dictionary of all lineshape settings, with keys corresponding to particle names or aliases, as {PARTICLE1: {'lineshape': 'NAME1', # E.g. "LSFLAT" or "LSNONRELBW"

'BlattWeisskopf': VALUE1, 'ChangeMassMin': VALUE12, 'ChangeMassMax': VALUE13},

PARTICLE2: {'lineshape': 'NAME2',
'BlattWeisskopf': VALUE2, 'ChangeMassMin': VALUE22, 'ChangeMassMax': VALUE23, 'IncludeBirthFactor': TRUE_OR_FALSE, 'IncludeDecayFactor': TRUE_OR_FALSE},

...

} where not all "sub-dictionaries" may contain all and/or the same keys.

dict_model_aliases() → dict[str, list[str]]

Return a dictionary of all model alias definitions in the input parsed file, of the form "ModelAlias <NAME> <MODEL>", as as {'NAME1': [MODEL_NAME, MODEL_OPTION1, MODEL_OPTION2,...],}.

dict_pythia_definitions() → dict[str, dict[str, str | float]]

Return a dictionary of all Pythia 8 commands "PythiaGenericParam" and/or "PythiaAliasParam" and/or "PythiaBothParam", with keys corresponding to the 3 types specifying whether the command is for generic decays, alias decays, or both. The commands are set in the input parsed file with statements of the form "Pythia<TYPE>Param <MODULE>:<PARAM>=<LABEL_OR_VALUE>. The dictionary takes the form {"PythiaAliasParam": {'<MODULE1>:<PARAM1>': 'LABEL1', '<MODULE1>:<PARAM2>': VALUE2, ...},

"PythiaBothParam": {'<MODULE2>:<PARAM3>': 'LABEL3', '<MODULE3>:<PARAM4>': VALUE4, ...}, "PythiaGenericParam": {'<MODULE4>:<PARAM5>': 'LABEL5', '<MODULE5>:<PARAM6>': VALUE6, ...}}.

expand_decay_modes(*particle: str*) → list[str]

Return a list of expanded decay descriptors for the given (mother) particle. The set of decay final states is effectively split and returned as a list. NB: this implicitly reverts aliases back to the original (EvtGen) names.

classmethod from_string(*filecontent: str*) → Self

Constructor from a .dec decay file provided as a multi-line string.

Parameters
filecontent (*str*) – Input .dec decay file content.

get_particle_property_definitions() → dict[str, dict[str, float]]

Return a dictionary of all particle property definitions in the input parsed file, of the form "Particle <PARTICLE> <MASS> <WIDTH>" or "Particle <PARTICLE> <MASS>", as {'PARTICLE1': {'mass': MASS1, 'width': WIDTH1},

'PARTICLE2': {'mass': MASS2, 'width': WIDTH2}, ...}.

Note:

1) Particles are often defined via aliases and post-processing may be needed to match the mass and width to the actual particle.
2) The mass (width) parameter is compulsory (optional). When not specified, the width is taken from the particle or alias.

---

global_photos_flag() → int

Return a boolean-like PhotosEnum enum specifying whether or not PHOTOS has been enabled.

Note: PHOTOS is turned on(off) for all decays with the global flag yesPhotos(noPhotos).

Returns
out (*PhotosEnum, default=PhotosEnum.no*) – PhotosEnum.yes / PhotosEnum.no if PHOTOS enabled / disabled

grammar() → str

Access the internal Lark grammar definition file, effectively loading the default grammar with default parsing options if no grammar has been loaded before.

Returns
out (*str*) – The Lark grammar definition file.

grammar_info() → dict[str, Any]

Access the internal Lark grammar definition file name and parser options, effectively loading the default grammar with default parsing options if no grammar has been loaded before.

Returns
out (*dict*) – The Lark grammar definition file name and parser options.

property grammar_loaded: bool

Check to see if the Lark grammar definition file is loaded.

list_charge_conjugate_decays() → list[str]

Return a (sorted) list of all charge conjugate decay definitions in the input parsed file, of the form "CDecay <MOTHER>", as ['MOTHER1', 'MOTHER2', ...].

list_decay_modes(*mother: str*, *pdg_name: bool = False*) → list[list[str]]

Return a list of decay modes for the given mother particle.

Parameters

- **mother** (*str*) – Input mother particle name.
- **pdg_name** (*bool, optional, default=False*) – Input mother particle name is the PDG name, not the (default) EvtGen name.

Example

```
>>> parser = DecFileParser('my-decay-file.dec')
>>> parser.parse()
>>> # Inspect what decays are defined
>>> parser.list_decay_mother_names()
>>> parser.list_decay_modes('pi0')
```

list_decay_mother_names() → list[str | Any]

Return a list of all decay mother names found in the parsed decay file.

list_lineshapePW_definitions() → list[tuple[list[str], int]]

Return a list of all SetLineshapePW definitions in the input parsed file, of the form "SetLineshapePW <MOTHER> <DAUGHTER1> <DAUGHTER2> <VALUE>", as [(['MOTHER1', 'DAUGHTER1-1', 'DAUGHTER1-2'], VALUE1), (['MOTHER2', 'DAUGHTER2-1', 'DAUGHTER2-2'], VALUE2), ...]

load_additional_decay_models(**models: str*) → None

Add one or more EvtGen decay models in addition to the ones already provided via `decaylanguage.dec.enums.known_decay_models`.

Parameters
models (*str*) – names of the additional decay models to be considered.

load_grammar(*filename: str | None = None*, *parser: str = 'lalr'*, *lexer: str = 'auto'*, ***options: Any*) → None

DEPRECATED, please use "`load_additional_decay_models`" instead.

Load a Lark grammar definition file, either the default one, or a user-specified one, optionally setting Lark parsing options.

Parameters

- **filename** (*str, optional, default=None*) – Input .dec decay file name. By default 'data/decfile.lark' is loaded.
- **parser** (*str, optional, default='lalr'*) – The Lark parser engine name.
- **lexer** (*str, optional, default='auto'*) – The Lark parser lexer mode to use.
- **options** (*keyword arguments, optional*) – Extra options to pass on to the parsing algorithm.
- **See Lark's Lark class for a description of available options**
- **for parser, lexer and options.**

`property number_of_decays: int`

Return the number of particle decays defined in the parsed .dec file.

`parse`(*include_ccdecays: bool = True*) → None

Parse the given .dec decay file(s) according to the default Lark parser and specified options.

Use the method `load_additional_decay_models` before `parse` to load decay models that might not yet be available in DecayLanguage.

Parameters

include_ccdecays (*boolean, optional, default=True*) – Choose whether or not to consider charge-conjugate decays, which are specified via "CDecay <MOTHER>". Make sure you understand the consequences of ignoring charge conjugate decays - you won't have a complete picture otherwise!

`print_decay_modes`(*mother: str*, *pdg_name: bool = False*, *print_model: bool = True*, *display_photos_keyword: bool = True*, *ascending: bool = False*, *normalize: bool = False*, *scale: float | None = None*) → None

Pretty print of the decay modes of a given particle, optionally with decay model information and/or normalisation or scaling of the branching fractions.

Parameters

- **mother** (*str*) – Input mother particle name.
- **pdg_name** (*bool, optional, default=False*) – Input mother particle name is the PDG name, not the (default) EvtGen name.
- **print_model** (*bool, optional, default=True*) – Specify whether to print the decay model and model parameters, if available.
- **display_photos_keyword** (*bool, optional, default=True*) – Display the "PHOTOS" keyword in decay models.
- **ascending** (*bool, optional, default=False*) – Print the list of decay modes ordered in ascending/descending order of branching fraction.
- **normalize** (*bool, optional, default=False*) – Print the branching fractions normalized to unity. The printing does not affect the values parsed and actually stored in memory.
- **scale** (*float | None, optional, default=None*) – If not None, the branching fractions (BFs) are normalized to the given value, which is taken to be the BF of the highest-BF mode of the list. Must be a number in the range]0, 1].

Examples

```
>>> s = '''Decay MyD_0*+
...  0.533   MyD0   pi+        PHSP;
...  0.08    MyD*0  pi+  pi0   PHSP;
...  0.0271  MyD*+  pi0  pi0   PHSP;
...  0.0542  MyD*+  pi+  pi-   PHSP;
... Enddecay
... '''
>>> p = DecFileParser.from_string(s)
>>> p.parse()
>>>
>>> # Simply print what has been parsed
>>> p.print_decay_modes("MyD_0*+")
  0.533             MyD0  pi+         PHSP;
  0.08              MyD*0 pi+ pi0     PHSP;
  0.0542            MyD*+ pi+ pi-     PHSP;
  0.0271            MyD*+ pi0 pi0     PHSP;
>>>
>>> # Print normalizing the sum of all mode BFs to unity
>>> p.print_decay_modes("MyD_0*+", normalize=True)
  0.7676796774      MyD0  pi+         PHSP;
  0.1152239666      MyD*0 pi+ pi0     PHSP;
  0.07806423736     MyD*+ pi+ pi-     PHSP;
  0.03903211868     MyD*+ pi0 pi0     PHSP;
>>>
>>> # Print scaling all BFs relative to the BF of the highest-BF mode in the␣
↪list,
>>> # the latter being set to the value of "scale".
>>> # In this example the decay file as printed would effectively signal, for␣
↪inspection,
>>> # that about 35% of the total decay width is not accounted for in the list␣
↪of modes,
>>> # since the sum of probabilities, interpreted as BFs, sum to about 65%.
>>> p.print_decay_modes("MyD_0*+", scale=0.5)
  0.5               MyD0  pi+         PHSP;
  0.07504690432     MyD*0 pi+ pi0     PHSP;
  0.05084427767     MyD*+ pi+ pi-     PHSP;
  0.02542213884     MyD*+ pi0 pi0     PHSP;
```

class decaylanguage.**DecayChain**(*mother: str*, *decays: dict[str,* DecayMode*]*)

Class holding a particle (single) decay chain, which is typically a top-level decay (mother particle, branching fraction and final-state particles) and a set of sub-decays for any non-stable particle in the top-level decay. The whole chain can be seen as a mother particle and a list of chained decay modes.

This class is the main building block for the digital representation of full decay chains.

Note: 1) Only single chains are supported, meaning every decaying particle can only define a single decay mode. 2) This class does not assume any kind of particle names (EvtGen, PDG). It is nevertheless advised to default use EvtGen names for consistency with the defaults used in the related classes `DecayMode` and `DaughtersDict`, unless there is a good motivation not to.

---

property bf: **float**

Branching fraction of the top-level decay.

flatten(*stable_particles: Iterable[dict[str, int] | list[str] | str] = ()*) → Self

Flatten the decay chain replacing all intermediate, decaying particles, with their final states.

Parameters

stable_particles (*iterable, optional, default=()*) – If provided, ignores the sub-decays of the listed particles, considering them as stable.

Note: After flattening the only `DecayMode` metadata kept is that of the top-level decay, i.e. that of the mother particle (nothing else would make sense).

Examples

```
>>> dm1 = DecayMode(0.0124, 'K_S0 pi0', model='PHSP')
>>> dm2 = DecayMode(0.692, 'pi+ pi-')
>>> dm3 = DecayMode(0.98823, 'gamma gamma')
>>> dc = DecayChain('D0', {'D0':dm1, 'K_S0':dm2, 'pi0':dm3})
>>>
>>> dc.flatten()
<DecayChain: D0 -> gamma gamma pi+ pi- (0 sub-decays), BF=0.008479803984>
>>> dc.flatten().to_dict()
{'D0': [{'bf': 0.008479803984,
   'fs': ['gamma', 'gamma', 'pi+', 'pi-'],
   'model': 'PHSP',
   'model_params': ''}]}
```

```
>>> dc.flatten(stable_particles=('K_S0', 'pi0')).decays
{'D0': <DecayMode: daughters=K_S0 pi0, BF=0.0124>}
```

classmethod from_dict(*decay_chain_dict: Dict[str, List[DecayModeDict]]*) → Self

Constructor from a decay chain represented as a dictionary. The format is the same as that returned by `DecFileParser.build_decay_chains(...)`.

property ndecays: int

Return the number of decay modes including the top-level decay.

print_as_tree() → None

Tree-structure like print of the entire decay chain.

Examples

```
>>> dm1 = DecayMode(0.028, 'K_S0 pi+ pi-')
>>> dm2 = DecayMode(0.692, 'pi+ pi-')
>>> dc = DecayChain('D0', {'D0':dm1, 'K_S0':dm2})
>>> dc.print_as_tree()
D0
+--> K_S0
|    +--> pi+
```

(continues on next page)

(continued from previous page)

```
|    +--> pi-
+--> pi+
+--> pi-
```

```
>>> dm1 = DecayMode(0.0124, 'K_S0 pi0')
>>> dm2 = DecayMode(0.692, 'pi+ pi-')
>>> dm3 = DecayMode(0.98823, 'gamma gamma')
>>> dc = DecayChain('D0', {'D0':dm1, 'K_S0':dm2, 'pi0':dm3})
>>> dc.print_as_tree()
D0
+--> K_S0
|    +--> pi+
|    +--> pi-
+--> pi0
     +--> gamma
     +--> gamma
```

```
>>> dm1 = DecayMode(0.6770, 'D0 pi+')
>>> dm2 = DecayMode(0.0124, 'K_S0 pi0')
>>> dm3 = DecayMode(0.692, 'pi+ pi-')
>>> dm4 = DecayMode(0.98823, 'gamma gamma')
>>> dc = DecayChain('D*+', {'D*+':dm1, 'D0':dm2, 'K_S0':dm3, 'pi0':dm4})
>>> dc.print_as_tree()
D*+
+--> D0
|    +--> K_S0
|    |    +--> pi+
|    |    +--> pi-
|    +--> pi0
|         +--> gamma
|         +--> gamma
+--> pi+
```

to_dict() → Dict[str, List[DecayModeDict]]

Return the decay chain as a dictionary representation. The format is the same as `DecFileParser.build_decay_chains(...)`.

Examples

```
>>> dm1 = DecayMode(0.028, 'K_S0 pi+ pi-')
>>> dm2 = DecayMode(0.692, 'pi+ pi-')
>>> dc = DecayChain('D0', {'D0':dm1, 'K_S0':dm2})
>>> dc.to_dict()
{'D0': [{'bf': 0.028,
    'fs': [{'K_S0': [{'bf': 0.692,
        'fs': ['pi+', 'pi-'],
        'model': '',
        'model_params': ''}]},
     'pi+',
     'pi-'],
```

(continues on next page)

(continued from previous page)

```
    'model': '',
    'model_params': ''}]}
```

to_string() → str

One-line string representation of the entire decay chain. Sub-decays are enclosed in round parentheses.

Examples

```
>>> dm1 = DecayMode(0.6770, "D0 pi+") # D*+
>>> dm2 = DecayMode(0.0124, "K_S0 pi0") # D0
>>> dm3 = DecayMode(0.692, "pi+ pi-") # K_S0
>>> dm4 = DecayMode(0.98823, "gamma gamma") # pi0
>>> dc = DecayChain("D*+", {"D*+":dm1, "D0":dm2, "K_S0":dm3, "pi0":dm4})
>>> print(dc.to_string())
D*+ -> (D0 -> (K_S0 -> pi+ pi-) (pi0 -> gamma gamma)) pi+
```

top_level_decay() → *DecayMode*

Return the top-level decay as a `DecayMode` instance.

property visible_bf: float

Visible branching fraction of the whole decay chain.

Note: Calculation requires a flattening of the entire decay chain.

---

class decaylanguage.**DecayChainViewer**(*decaychain: dict[str, list[dict[str, float | str | list[Any]]]]*, ***attrs: dict[str, bool | int | float | str]*)

The class to visualize a decay chain.

Examples

```
>>> dfp = DecFileParser('my-Dst-decay-file.dec')
>>> dfp.parse()
>>> chain = dfp.build_decay_chains('D*+')
>>> dcv = DecayChainViewer(chain)
>>> # display the SVG figure in a notebook
>>> dcv
```

When not in notebooks the graph can easily be visualized with the `graphviz.Digraph.render` or `graphviz.Digraph.view` functions, e.g.: >>> dcv.graph.render(filename="test", format="pdf", view=True, cleanup=True) # doctest: +SKIP

property graph: Digraph

Get the actual `graphviz.Digraph` object. The user now has full control ...

to_string() → str

Return a string representation of the built graph in the DOT language. The function is a trivial shortcut for `graphviz.Digraph.source`.

class decaylanguage.**DecayMode**(*bf: float = 0*, *daughters:* DaughtersDict | *dict[str, int]* | *list[str]* | *tuple[str]* | *str* | *None = None*, ***info: Any*)

Class holding a particle decay mode, which is typically a branching fraction and a list of final-state particles (i.e. a list of DaughtersDict instances). The class can also contain metadata such as decay model and optional decay-model parameters, as defined for example in .dec decay files.

This class is a building block for the digital representation of full decay chains.

Note: This class assumes EvtGen particle names, though this assumption is only relevant for the `charge_conjugate` method. Otherwise, all other class methods smoothly deal with any kind of particle names (basically an iterable of strings).

---

charge_conjugate(*pdg_name: bool = False*) → Self

Return the charge-conjugate decay mode.

Parameters

pdg_name (*str, optional, default=False*) – Input particle name is the PDG name, not the (default) EvtGen name.

Examples

```
>>> dm = DecayMode(1.0, 'K+ K+ pi-')
>>> dm.charge_conjugate()
<DecayMode: daughters=K- K- pi+, BF=1.0>
>>>
>>> dm = DecayMode(1.0, 'K_S0 pi+')
>>> dm.charge_conjugate()
<DecayMode: daughters=K_S0 pi-, BF=1.0>
>>>
>>> dm = DecayMode(1.0, 'K(S)0 pi+')  # PDG names!
>>> dm.charge_conjugate(pdg_name=True)
<DecayMode: daughters=K(S)0 pi-, BF=1.0>
```

describe() → str

Make a nice high-density string for all decay-mode properties and info.

classmethod from_dict(*decay_mode_dict: DecayModeDict*) → Self

Constructor from a dictionary of the form {'bf': <float>, 'fs': [...], ...}. These two keys are mandatory. All others are interpreted as model information or metadata, see the constructor signature and doc.

Note: This class assumes EvtGen particle names, though this assumption is only relevant for the `charge_conjugate` method. Otherwise, all other class methods smoothly deal with any kind of particle names (basically an iterable of strings).

Examples

```
>>> # Simplest construction
>>> DecayMode.from_dict({'bf': 0.98823, 'fs': ['gamma', 'gamma']})
<DecayMode: daughters=gamma gamma, BF=0.98823>
```

```
>>> # Decay mode with decay model details
>>> DecayMode.from_dict({'bf': 0.98823,
...                      'fs': ['gamma', 'gamma'],
...                      'model': 'PHSP',
...                      'model_params': ''})
<DecayMode: daughters=gamma gamma, BF=0.98823>
```

```
>>> # Decay mode with metadata for generators such as zfit's phasespace
>>> dm = DecayMode.from_dict({'bf': 0.5, 'fs': ["K+, K-"], "zfit": {"B0": "gauss
↪"}})
>>> dm.metadata
{'model': '', 'model_params': '', 'zfit': {'B0': 'gauss'}}
```

classmethod from_pdgids(*bf: float = 0*, *daughters: list[int] | tuple[int] | None = None*, ***info: Any*) → Self

Constructor for a final state given as a list of particle PDG IDs.

Parameters

- **bf** (*float, optional, default=0*) – Decay mode branching fraction.
- **daughters** (*list or tuple, optional, default=None*) – The final-state particle PDG IDs.
- **info** (*keyword arguments, optional*) – Decay mode model information and/or user metadata (aka extra info) By default the following elements are always created: dict(model=None, model_params=None). The user can provide any metadata, see the examples below.

Note: All particle names are internally saved as EvtGen names, to be consistent with the default class assumption, see class docstring.

Examples

```
>>> DecayMode.from_pdgids(0.5, [321, -321])
<DecayMode: daughters=K+ K-, BF=0.5>
```

```
>>> DecayMode.from_pdgids(0.5, (310, 310))
<DecayMode: daughters=K_S0 K_S0, BF=0.5>
```

```
>>> # Decay mode with metadata
>>> dm = DecayMode.from_pdgids(0.5, (310, 310), model="PHSP")
>>> dm.metadata
{'model': 'PHSP', 'model_params': ''}
```

to_dict() → dict[str, int | float | str | list[str]]

Return the decay mode as a dictionary in the format understood by the `DecayChainViewer` class.

Examples

```
>>> dm = DecayMode(0.5, 'K+ K- K- pi- pi0 nu_tau', model='PHSP', study='toy',␣
↪year=2019)
>>> dm.to_dict()
{'bf': 0.5,
 'fs': ['K+', 'K-', 'K-', 'nu_tau', 'pi-', 'pi0'],
 'model': 'PHSP',
 'model_params': '',
 'study': 'toy',
 'year': 2019}
```

3.2 decaylanguage.decay

3.2.1 decaylanguage.modeling.decay

A general base class representing decays.

class decaylanguage.modeling.decay.**ModelDecay**(*particle*, *daughters=[]*, *name=None*)

This describes a decay very generally, with search and print features. Subclassed for further usage.

list_structure(*final_states*)

The structure in the form [(0,1,2,3)], where the dual-list is used for permutations for bose symmatrization. So for final_states=[a,b,c,c], [a,c,[c,b]] would be: [(0,2,3,1),(0,3,2,1)]

property structure

The structure of the decay chain, simplified to only final state particles

3.2.2 decaylanguage.modeling.ampgentransform

class decaylanguage.modeling.ampgentransform.**AmpGenTransformer**(*visit_tokens: bool = True*)

3.2.3 decaylanguage.modeling.amplitudechain

A class representing a set of decays. Can be subclassed to provide custom converters.

class decaylanguage.modeling.amplitudechain.**AmplitudeChain**(*particle*, *daughters=[]*, *lineshape=None*, *spinfactor=None*, *amp=1 + 0j*, *err=0j*, *fix=True*, *name=None*)

This is a chain of decays (a "line")

expand_lines(*linelist*)

Take a DecayTree -> list of DecayTrees with each dead-end daughter expanded to every possible combination. (recursive)

classmethod from_matched_line(*mat*)

This operates on an already-matched line.

Parameters

mat – The groupdict output of a match

Returns
A new amplitude chain instance

classmethod read_ampgen(*filename=None*, *text=None*, *grammar=None*, *parser='lalr'*, ***kargs*)

Read in an ampgen file

Parameters

- **filename** – Filename to read
- **text** – Text to read (use instead of filename)

Returns
array of AmplitudeChains, parameters, constants, event type

class decaylanguage.modeling.amplitudechain.**LS**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Line shapes supported (currently)

3.2.4 decaylanguage.modeling.goofit

This is a GooFit adaptor for amplitude chain.

class decaylanguage.modeling.goofit.**DecayStructure**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

class decaylanguage.modeling.goofit.**GooFitChain**(*particle*, *daughters=[]*, *lineshape=None*, *spinfactor=None*, *amp=1 + 0j*, *err=0j*, *fix=True*, *name=None*)

Class to read an AmpGen options file and return GooFit C++ code.

property decay_structure

Determine if decay proceeds via two resonances or cascade decay.

property formfactor

Return form factor based on relative angular momentum L.

make_amplitude(*final_states*)

Write out the amplitude and push it to a vector.

classmethod make_intro(*all_states*)

Write out definitions of constant variables and vectors to hold intermediate values.

make_linefactor(*final_states*)

Write out the line shape and push it to a vector.

make_lineshape(*structure*, *masses*)

Write out the line shapes. Each kind of line shape is treated separately.

classmethod make_pars()

Write out the parameters used in the amplitudes.

make_spinfactor(*final_states*)

Write out the spin factor and push it to a vector.

classmethod read_ampgen(**args*, ***kargs*)

Read in an AmpGen file.

Returns
Array of AmplitudeChains, event type.

spindetails()

Return string with spin structure.

property spinfactors

Check if the spin structure is known and return it together with the form factor.

to_goofit(*final_states*)

Write the vectors with the spin factors, line shapes and amplitudes.

class decaylanguage.modeling.goofit.**GooFitPyChain**(*particle*, *daughters=[]*, *lineshape=None*, *spinfactor=None*, *amp=1 + 0j*, *err=0j*, *fix=True*, *name=None*)

Class to read an AmpGen options file and return a GooFit Python script.

property decay_structure

Determine if the decay proceeds via two resonances or a cascade decay.

property formfactor

Return the form factor based on relative angular momentum L.

make_amplitude(*final_states*)

Write out the amplitude and push it to a vector.

classmethod make_intro(*all_states*)

Write out definitions of constant variables and lists to hold intermediate values.

make_linefactor(*final_states*)

Write out the line shape and push it to a vector.

make_lineshape(*structure*, *masses*)

Write out the line shapes. Each kind of line shape is treated separately.

classmethod make_pars()

Write out the parameters used in the amplitudes.

make_spinfactor(*final_states*)

Write out the spin factor and push it to a vector.

classmethod read_ampgen(**args*, ***kargs*)

Read in the AmpGen file.

Returns
Array of AmplitudeChains, event type.

spindetails()

Return string with spin structure.

property spinfactors

Check if the spin structure is known and return it together with the form factor.

to_goofit(*final_states*)

Write the lists with the spin factors, line shapes and amplitudes.

class decaylanguage.modeling.goofit.**SF_4Body**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When ``reporting a bug <https://github.com/scikit-hep/decaylanguage/issues>``_ please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

decaylanguage could always use more documentation, whether as part of the official decaylanguage docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/scikit-hep/decaylanguage/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

To set up `decaylanguage` for local development:

1. Fork ``decaylanguage <https://github.com/scikit-hep/decaylanguage>``_ (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/decaylanguage.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with ``nox <https://nox.thea.codes/en/stable/>``_ one command:

```
nox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `nox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.md` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
nox -s tests-3.9 -- -k test_myfeature
```

[1] If you don't have all the necessary python versions available locally, you can run a specific nox session with `nox -s tests-3.9`, for example. GitHub Actions will run all the tests whenever a pull request is made, so testing all versions of Python locally is usually not necessary.

CHAPTER

FIVE

AUTHORS

- Henry Fredrick Schreiner III - https://iscinumpy.gitlab.io
- Eduardo Rodrigues - https://github.com/eduardo-rodrigues/

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

G

L

M

N

P

R

S

T

V