
DYCI2 library documentation

Release 0.1

Jérôme Nika, IRCAM STMS LAB

Oct 17, 2017

TABLE OF CONTENTS

1	Introduction	1
2	Overview	3
3	Models and navigation	7
3.1	Model	7
3.2	Navigator	13
3.3	Meta Model Navigator	18
3.4	Factor Oracle Navigator	19
4	Handle generation	23
4.1	Query	23
4.2	Generator	24
5	Pattern matching algorithms	31
5.1	Prefix indexing algorithms	31
5.2	Using Intervals	34
6	Event, Label, Transformation	35
6.1	Label	35
6.2	Transforms	35
7	OSC communication	37
7.1	OSC AGENT	37
8	Memory and corpus	39
8.1	Generator Builder	39
9	Indices and tables	45
	Python Module Index	47
	Index	49

INTRODUCTION

This version of the *DYCI2 library* is the first release of a work in progress: a set of models and tools for creative generation of sequences (and in particular musical sequences) from models of sequences. It implements several models, generative heuristics, time management strategies, and architectures of interactive agents.

DYCI2 (“Creative Dynamics of Improvised Interaction”) is a collaborative research and development project funded by the French National Research Agency (ANR). It explores the creative dynamics of improvised interactions between human and artificial agents, featuring an informed artificial listening scheme, a musical structure discovery and learning scheme, and a generalized interaction / knowledge / decision dynamics scheme (see <http://repmus.ircam.fr/dyci2/home>). The *DYCI2 library* is part of the DYCI2 project, it is conceived as an autonomous and easily extensible Python library, and can also be used in association with audio or midi listeners and renderers to form *DYCI2 agents* (see directory “*MaxPatches*”).

More information on the project: refer to **Nika, Déguernel, Chemla–Romeu-Santos, Vincent, Assayag, “DYCI2 agents: merging the “free”, “reactive”, and “scenario-based” music generation paradigms”, in Proceedings of International Computer Music Conference 2017** (<https://hal.archives-ouvertes.fr/hal-01583089/document>) (please mention this paper to mention the library).

To use the library: see the tutorials in the root directory.

In this version:

- Definitive architecture of the library: classes Model, Navigator, MetaModelNavigator, Query, Generator, GenerationHandler, and OSCAgent.
- *Models*: Naïve sequence navigation, Factor Oracle automaton.
- *Associated navigation strategies*: free generation (“Omax-like”), single target generation, scenario-based generation (“ImproteK-like”).
- *Architectures of generative agent*: Generator and GenerationHandler (see doc), demand-driven generative agents processing queries expressed in “events”. The (short-term and long-term) queries communicate through a shared execution trace to maintain consistency of the generated sequence when rewriting previously generated anticipations.
- *Communication*: OSC agents embedding the generative agents.
- *Alphabets*: Generic list labels, Chord Labels.
- *Documentation*: modules Model, Navigator, PrefixIndexing, MetaModelNavigator, Generator, GeneratorBuilder.
- *Tutorials*: FactorOracle, PrefixIndexing, Intervals, MetaModelNavigator, FactorOracleNavigator, Generator, GenerationHandler, OSCAgent_Tutorial_1 (text).

Next steps:

- *Models*: N-gram.

- *Associated navigation strategies*: Navigator including a notion of “activity profile”, free generation and single target generation using this navigator (“Somax-like”).
- *Architectures of generative agent*: “Algebra” of queries, queries expressed in “ms”. output of the queries not directly outputted but redirected to a choice method that will be in charge of outputting the result → new playing modes “hard hybrid guidance” (long-term queries > short-term queries) and “soft hybrid guidance” (short-term queries > long-term queries).
- *Alphabets*: Class “contents”.
- *Documentation*: modules Label, Transforms, Intervals, ParseAnnotationFiles, CorpusBuilder, OSCAgent.
- *Tutorials*: GeneratorBuilder, OSCAgent_Tutorial_2 (audio), OSCAgent_Tutorial_3 (midi).

OVERVIEW

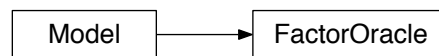


Fig. 2.1: *Model* is an abstract class. Any new model of sequence must inherit from this class. The classes inheriting from this class are minimal and only implement the construction algorithms and basic methods. Navigation and creative aspects are handled by the classes introduced below. Models of sequence implemented so far: Factor Oracle Automaton (*FactorOracle*).

```
graph TD; Navigator[Navigator]
```

The diagram shows a single box labeled 'Navigator'.

Fig. 2.2: The class *Navigator* implements parameters and methods that are used to navigate through a model of sequence. These parameters and methods are **model-independent**. This class defines in particular the naive versions of the methods *simply_guided_generation()* and *free_generation()* handling the navigation through a sequence when it is respectively guided by target labels and free. These methods are overloaded by model-dependant versions (and other model-dependent parameters or methods can be added) when creating a **model navigator** class (cf. below).



```
graph TD; MetaModelNavigator[MetaModelNavigator];
```

Fig. 2.3: *MetaModelNavigator* is a **metaclass**. A new class created using this metaclass is a **model navigator** and inherits from: **1)** a class inheriting from *Model*, **2)** a class inheriting from *Navigator*. A **model navigator** implements the different algorithms, strategies, and heuristics to navigate through a given model of sequence for analysis or creative applications, for example **generating new sequences using concatenative synthesis of events learned in the model**. The class *FactorOracleNavigator* introduced below is an example of model navigator created using this metaclass.

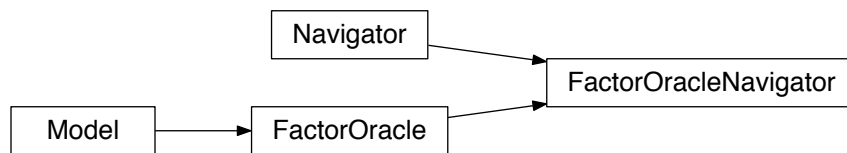


Fig. 2.4: The class *FactorOracleNavigator* implements different algorithms, strategies, and heuristics to navigate through a Factor Oracle Automaton for creative applications. The creation of the class *FactorOracleNavigator* in the file *ModelNavigator.py* gives an example of easy definition of a new class of model navigator using the metaclass *MetaModelNavigator*: **1)** chose two bases (here, `model = FactorOracle`, `navigator = Navigator`), **2)** define the methods to overload `simply_guided_generation()` and `free_generation()` (here `ModelNavigator.FactorOracleNavigator.simply_guided_navigation()` and `ModelNavigator.FactorOracleNavigator.free_navigation()`)



Fig. 2.5: The class *Generator* embeds a **model navigator** (cf. metaclass *MetaModelNavigator*) as “memory” and processes **queries** (class *Query*) to generate new sequences. This class uses pattern matching techniques (cf. *PrefixIndexing*) to enrich the navigation and generation methods offered by the chosen model. The class *GenerationHandler* introduces time management and planning for interactive applications and adds a pool of query, concurrency (e.g. processing of concurrent queries), etc. to the class *Generator*.

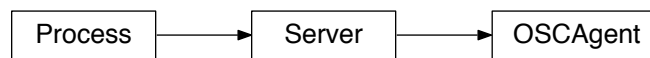


Fig. 2.6: The class `Server` defines an OSC (<http://opensoundcontrol.org>) server to communicate with external applications. The class `OSCAgent` defines a process embedding an instance of `GenerationHandler` and its own OSC sender, receiver, settings, etc. to receive queries and control parameters and send generation outputs. Both classes can be used in particular in association with a Max interface handling audio or midi rendering.

MODELS AND NAVIGATION

3.1 Model

This module defines different models of symbolic sequences. The classes defined in this module are minimal and only implement the construction algorithms and basic methods. Navigation and creative aspects are handled by other classes in the library (cf. *Navigator* and *ModelNavigator*). Main classes: *Model*, *FactorOracle*. Tutorial for the class *FactorOracle* in `_Tutorials_/FactorOracleAutomaton_tutorial.py`.

class `Model.FactorOracle` (*sequence*=[], *labels*=[], *equiv*=<function <lambda>>)

Bases: `Model.Model`

Factor Oracle automaton class. Implementation of the Factor Oracle Automaton (Allauzen, Crochemore, Raffinot, 1999). Convention: since the all the transitions arriving in a same state have the same label, the labels are not carried by the transitions but by the states.

Parameters

- **sequence** (*list* or *str*) – sequence learnt in the Factor Oracle automaton
- **labels** (*list* or *str*) – sequence of labels chosen to describe the sequence
- **direct_transitions** (*dict*) – direct transitions in the automaton (key = index state 1, value = tuple: label, index state 2)
- **factor_links** (*dict*) – factor links in the automaton (key = index state 1, value = list of tuples: (label, index state 2))
- **suffix_links** (*dict*) – suffix links in the automaton (key = index state 1, value = index state 2)
- **reverse_suffix_links** (*dict*) – reverse suffix links in the automaton (key = index state 1, value = list of index state 2)
- **equiv** (*function*) – comparison function given as a lambda function, default: (lambda x,y : x == y).

! **equiv** has to be consistent with the type of the elements in labels.

See also Tutorial in `_Tutorials_/FactorOracleAutomaton_tutorial.py`.

(When there is no need to distinguish the sequence and its labels : `FactorOracle(sequence,sequence)`.)

Example

```
>>> sequence = ['A', 'B', 'B', 'C', 'A', 'B', 'C', 'D', 'A', 'B', 'C']
>>> FO = FactorOracle(sequence, sequence)
>>>
>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
```

```
>>> labels = [s[0] for s in sequence]
>>> FO_2 = FactorOracle(sequence, labels)
>>>
>>> equiv_AC_BD = (lambda x,y: set([x,y]).issubset(set(['A','C'])) or set([x,y]).
↳issubset(set(['B','D'])))
>>> FO_3 = FactorOracle(sequence, labels, equiv_AC_BD)
```

add_direct_transition (*index_state1*, *label*, *index_state2*)

Adds a transition labelled by ‘label’ from the state at index ‘index_state1’ to the state at index ‘index_state2’ in the Factor Oracle automaton.

add_factor_link (*index_state1*, *label*, *index_state2*)

Adds a factor link labelled by ‘label’ from the state at index ‘index_state1’ to the state at index ‘index_state2’ in the Factor Oracle automaton.

add_suffix_link (*index_state1*, *index_state2*)

Adds a suffix link (and the associated reverse suffix link) from the state at index ‘index_state1’ to the state at index ‘index_state2’ in the Factor Oracle automaton.

continuations (*index_state*, *forward_context_length_min*=1, *equiv*=None, *authorize_direct_transition*=True)

Possible continuations from the state at index *index_state* in the automaton, i.e. direct transition and states reached using suffix links and reverse suffix links. These states follow states sharing a common backward context and a common forward context with the state at index *index_state* in the automaton. The lengths of the common backward contexts are given by the Factor Oracle automaton, the forward context is imposed by a parameter.

Parameters

- **index_state** (*int*) – start index
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **equiv** (*function*) – Comparison function given as a lambda function, default: self.equiv.
- **authorize_direct_transition** (*bool*) – include direct transitions ?

Returns Indexes in the automaton of the possible continuations from the state at index *index_state* in the automaton.

Return type list (*int*)

See also **Tutorial in** `_Tutorials_/FactorOracleAutomaton_tutorial.py`.

! **equiv** has to be consistent with the type of the elements in labels.

Example

```
>>> sequence = ['A1','B1','B2','C1','A2','B3','C2','D1','A3','B4','C3']
>>> labels = [s[0] for s in sequence]
>>> FON = FactorOracleNavigator(sequence, labels)
>>>
>>> index = 6
>>> forward_context_length_min = 1
>>> continuations = FON.continuations(index, forward_context_length_min)
>>> print("Possible continuations from state at index {} (with minimum_
↳forward context length = {}): {}".format(index, forward_context_length_min,
↳continuations))
```

continuations_with_jump (*authorized_indexes*)

List of continuations with jumps to indexes with similar contexts direct transition from `self.current_navigation_index`.

In the method `free_generation`, this method is called with `authorized_indexes` = possible continuations filtered to satisfy the constraints of taboos and repetitions. In the method `simply_guided_generation`, this method is called with `authorized_indexes` = possible continuations **matching the required label** filtered to satisfy the constraints of taboos and repetitions.

Parameters `authorized_indexes` (*list(int)*) – list of authorized indexes to filter taboos, repetitions, and label when needed.

Returns indexes of the states

Return type `list(int)`

continuations_with_label (*index_state*, *required_label*, *forward_context_length_min=1*, *equiv=None*, *authorize_direct_transition=True*)

Possible continuations labeled by `required_label` from the state at index `index_state` in the automaton.

Parameters

- `index_state` (*int*) – start index
- `required_label` – label to read
- `forward_context_length_min` (*int*) – minimum length of the forward common context
- `equiv` (*function*) – Comparison function given as a lambda function, default: `self.equiv`.
- `authorize_direct_transition` (*bool*) – include direct transitions?

Returns Indexes in the automaton of the possible continuations labeled by `required_label` from the state at index `index_state` in the automaton.

Return type `list(int)`

See also method `from_state_read_label` (class `FactorOracle`) used in the construction algorithm.
Difference : only uses the direct transition and the suffix link leaving the state.

! `equiv` has to be consistent with the type of the elements in labels.

follow_continuation_using_transition (*authorized_indexes*)

Continuation using direct transition from `self.current_navigation_index`.

In the method `free_generation`, this method is called with `authorized_indexes` = possible continuations filtered to satisfy the constraints of taboos and repetitions. In the method `simply_guided_generation`, this method is called with `authorized_indexes` = possible continuations **matching the required label** filtered to satisfy the constraints of taboos and repetitions.

Parameters `authorized_indexes` (*list(int)*) – list of authorized indexes to filter taboos, repetitions, and label when needed.

Returns index of the state

Return type `int`

follow_continuation_with_jump (*authorized_indexes*)

Random selection of a continuation with jump to indexes with similar contexts direct transition from `self.current_navigation_index`.

In the method `free_generation`, this method is called with `authorized_indexes` = possible continuations filtered to satisfy the constraints of taboos and repetitions. In the method `simply_guided_generation`, this

method is called with `authorized_indexes = possible continuations matching the required label` filtered to satisfy the constraints of taboos and repetitions.

Parameters `authorized_indexes` (`list(int)`) – list of authorized indexes to filter taboos, repetitions, and label when needed.

Returns index of the state

Return type `int`

follow_reverse_suffix_links_from (`index_state`)

Reverse suffix paths from a given index.

Parameters `index_state` (`int`) – start index

Returns Indexes in the automaton of the states that can be reached from the state at index `index_state` following reverse suffix links.

Return type `list(int)`

follow_suffix_links_from (`index_state`, `include_init_state=True`)

Suffix path from a given index.

Parameters `index_state` (`int`) – start index

Returns Indexes in the automaton of the states that can be reached from the state at index `index_state` following suffix links.

Return type `list(int)`

follow_suffix_links_then_reverse_from (`index_state`)

States that can be reached using suffix links from the state at index `index_state`, and then the reverse suffix links leaving these states.

Parameters `index_state` (`int`) – start index

Returns Indexes in the automaton of the states that can be reached using suffix links from the state at index `index_state`, and then the reverse suffix links leaving these states.

Return type `list(int)`

from_state_read_label (`index_state`, `label`, `equiv=None`, `authorize_factor_links=True`)

Reads label ‘label’ from state at index ‘index_state’. First looks for a direct transition, then for a factor link (if authorized).

Parameters

- **index_state** (`int`) – Initial state in the Factor Oracle automaton.
- **label** – Label to read.
- **equiv** (`function`) – Comparison function given as a lambda function, default if no parameter is given: `self.equiv`.
- **authorize_factor_links** (`bool`) – Only look for a direct transition (False) or also for a factor link (True).

Returns Index where the transition leads (when it exists).

Return type `int`

init_model ()

Creation of the initial state of the Factor Oracle automaton. (“Empty”, no label, suffix links going “nowhere”)

is_recognized (*word*, *equiv=None*)

Tests if a word is recognized by the Factor Oracle automaton.

Parameters

- **word** (*list* or *str*) – Input sequence
- **equiv** (*function*) – Compararison function given as a lambda function, default if no parameter is given: self.equiv.

! **equiv** has to be consistent with the type of label.

See also **Tutorial in** `_Tutorials_/FactorOracleAutomaton_tutorial.py`.

Example

```
>>> sequence_FO = "AABBABCBBABAAB"
>>> FO = FactorOracle(sequence_FO, sequence_FO)
>>> sequence_input_1 = "ABCB"
>>> sequence_input_2 = "BBBBBB"
>>> print("{} recognized by the Factor Oracle built on {}?\n{}".
↪format(sequence_input_1, sequence_FO, FO.is_recognized(sequence_input_1)))
>>> print("{} recognized by the Factor Oracle built on {}?\n{}".
↪format(sequence_input_2, sequence_FO, FO.is_recognized(sequence_input_2)))
```

learn_event (*state*, *label*, *equiv=None*)

Learns (appends) a new state in the Factor Oracle automaton.

Parameters

- **state** –
- **label** –
- **equiv** (*function*) – Compararison function given as a lambda function, default if no parameter is given: self.equiv.

! **equiv** has to be consistent with the type of label.

print_model ()

Basic representation of a Factor Oracle automaton.

similar_backward_context (*index_state*)

Some states sharing a common (backward) context with the state at index *index_state* in the automaton.

Parameters *index_state* (*int*) – start index

Returns Indexes in the automaton of the states sharing a common (backward) context with the state at index *index_state* in the automaton.

Return type list (*int*)

See also <https://hal.archives-ouvertes.fr/hal-01161388>

See also **Tutorial in** `_Tutorials_/FactorOracleAutomaton_tutorial.py`.

Example

```
>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels = [s[0] for s in sequence]
>>> FON = FactorOracleNavigator(sequence, labels)
>>>
>>> index = 6
>>> similar_backward_context = FON.similar_backward_context(index)
>>> print("Some states with backward context similar to that of state at_
↪index {}: {}".format(index, similar_backward_context))
```

similar_contexts (*index_state*, *forward_context_length_min*=1, *equiv*=None)

Some states sharing a common backward context and a common forward context with the state at index *index_state* in the automaton. The lengths of the common backward contexts are given by the Factor Oracle automaton, the forward context is imposed by a parameter.

Parameters

- **index_state** (*int*) – start index
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **equiv** (*function*) – Compararison function given as a lambda function, default: self.equiv.

Returns Indexes of the states in the automaton sharing a common backward context and a common forward context with the state at index *index_state* in the automaton.

Return type list (*int*)

See also Tutorial in `_Tutorials_/FactorOracleAutomaton_tutorial.py`.

! **equiv** has to be consistent with the type of the elements in labels.

Example

```
>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels = [s[0] for s in sequence]
>>> FON = FactorOracleNavigator(sequence, labels)
>>>
>>> index = 6
>>> forward_context_length_min = 1
>>> similar_contexts = FON.similar_contexts(index, forward_context_length_min)
>>> print("Some states with similar contexts (with minimum forward context_
↪length = {}) to that of state at index {}: {}".format(forward_context_
↪length_min, index, similar_contexts))
```

class `Model.Model` (*sequence*=[], *labels*=[], *equiv*=<function <lambda>>)

Bases: `object`

The class `Model` is an **abstract class**. Any new model of sequence must inherit from this class.

Parameters

- **sequence** (*list* or *str*) – sequence learnt in the model.
- **labels** (*list* or *str*) – sequence of labels chosen to describe the sequence.
- **equiv** (*function*) – compararison function given as a lambda function, default if no parameter is given: self.equiv.

! **equiv** has to be consistent with the type of the elements in labels.

build (*sequence*, *labels*, *equiv*=None)

Builds the model.

Parameters

- **sequence** (*list* or *str*) – sequence learnt in the model.
- **labels** (*list* or *str*) – sequence of labels chosen to describe the sequence

- **equiv** (*function*) – Compararison function given as a lambda function, default if no parameter is given: `self.equiv`.

! **equiv** has to be consistent with the type of the elements in labels.

index_last_state ()

Index of the last state in the model.

init_model ()

Initialization method called before learning the sequence in the model.

learn_event (*state, label, equiv*)

Learns (appends) a new state in the model.

Parameters

- **state** –
- **label** –
- **equiv** (*function*) – Compararison function given as a lambda function, default if no parameter is given: `self.equiv`.

! **equiv** has to be consistent with the type of label.

learn_sequence (*sequence, labels, equiv=None*)

Learns (appends) a new sequence in the model.

Parameters

- **sequence** (*list or str*) – sequence learnt in the Factor Oracle automaton
- **labels** (*list or str*) – sequence of labels chosen to describe the sequence
- **equiv** (*function*) – Compararison function given as a lambda function, default if no parameter is given: `self.equiv`.

! **equiv** has to be consistent with the type of the elements in labels.

print_model ()

3.2 Navigator

This module defines parameters and methods to navigate through a symbolic sequence. The classes defined in this module are used in association with models (cf. [Model](#)) when creating **model navigator** classes (cf. [ModelNavigator](#)).

```
class Navigator.Navigator (sequence=[], labels=[], max_continuity=2, control_parameters=[], execution_trace_parameters=[], equiv=<function <lambda>>)
```

Bases: `object`

The class `Navigator` implements **parameters and methods that are used to navigate through a model of sequence**. These parameters and methods are **model-independent**. This class defines in particular the naive versions of the methods `Navigator.simply_guided_navigation()` and `Navigator.free_navigation()` handling the navigation through a sequence when it is respectively guided by target labels and free. These methods are overloaded by model-dependant versions (and other model-dependent parameters or methods can be added) when creating a **model navigator** class (cf. [ModelNavigator](#)). This class is not supposed to be used alone, only in association with a model within a model navigator. Therefore its attributes are only “flags” that can be used when defining a model navigator.

Parameters

- **sequence** (*list or str*) – sequence learnt in the model.
- **labels** (*list or str*) – sequence of labels chosen to describe the sequence.
- **equiv** (*function*) – comparison function given as a lambda function, default if no parameter is given: `self.equiv`.
- **current_navigation_index** (*int*) – current length of the navigation
- **current_position_in_sequence** (*int*) – current position of the readhead in the model. ****** When this attribute receives a new value, `Navigator.record_execution_trace()` is called to update `self.execution_trace`, and `Navigator.update_history_and_taboos()` is called to update `self.history_and_taboos`.******
- **current_continuity** (*int*) – current number of consecutive elements retrieved in the sequence at the current step of generation
- **max_continuity** (*int*) – limitation of the length of the sub-sequences that can be retrieved from the sequence.
- **no_empty_event** (*bool*) – authorize or not to output empty events.
- **avoid_repetitions_mode** (*int*) – 0: authorize repetitions; 1: favor less previously retrieved events; 2: forbid repetitions.
- **control_parameters** (*list (str)*) – list of the slots of the class that are considered as “control parameters” i.e. that can be used by a user to author / monitor the generation processes.
- **execution_trace_parameters** – list of the slots of the class that are stored in the execution trace used in `Generator.go_to_anterior_state_using_execution_trace()`.
- **execution_trace** (*dict*) – History of the previous runs of the generation model. The list of the parameters of the model whose values are stored in the execution trace is defined in `self.execution_trace_parameters`.

authorize_indexes (*indexes*)

Delete the “taboos” (events that cannot be retrieved) in the navigation mechanisms for the states listed in the parameter indexes.

Parameters indexes (*list (int)*) – indexes of authorized indexes (!depending on the model the first event can be at index 0 or 1).

delete_taboos ()

Delete all the “taboos” (events that cannot be retrieved) in the navigation mechanisms.

filter_using_history_and_taboos (*list_of_indexes*)

find_matching_label_without_continuation (*required_label, authorized_indexes, equiv=None*)

Random state in the sequence matching `required_label` if `self.no_empty_event` is `True` (else `None`).

Parameters

- **required_label** – label to read
- **authorized_indexes** (*list (int)*) – list of authorized indexes to filter taboos, repetitions, and label when needed.
- **equiv** (*function*) – Comparison function given as a lambda function, default: `self.equiv`.

Returns index of the state

Return type `int`

! **equiv** has to be consistent with the type of the elements in labels.

forbid_indexes (*indexes*)

Introduces “taboos” (events that cannot be retrieved) in the navigation mechanisms.

Parameters **indexes** (*list(int)*) – indexes of forbidden indexes (!depending on the model the first event can be at index 0 or 1).

free_generation (*length, new_max_continuity=None, forward_context_length_min=0, init=False, equiv=None, print_info=False*)

Free navigation through the sequence. Naive version of the method handling the free navigation in a sequence (random). This method has to be overloaded by a model-dependant version when creating a **model navigator** class (cf. `ModelNavigator`).

Parameters

- **length** (*int*) – length of the generated sequence
- **new_max_continuity** (*int*) – new value for `self.max_continuity` (not changed if no parameter is given)
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **init** (*bool*) – reinitialise the navigation parameters ?, default : `False`. (True when starting a new generation)
- **equiv** (*function*) – Comparison function given as a lambda function, default: `self.equiv`.
- **print_info** (*bool*) – print the details of the navigation steps

Returns generated sequence

Return type `list`

See also Example of overloaded method: `FactorOracleNavigator.free_navigation()`.

! **equiv** has to be consistent with the type of the elements in labels.

! The result **strongly depends** on the tuning of the parameters `self.max_continuity`, `self.avoid_repetitions_mode`, `self.no_empty_event`.

free_navigation (*length, new_max_continuity=None, forward_context_length_min=0, init=False, equiv=None, print_info=False*)

Free navigation through the sequence. Naive version of the method handling the free navigation in a sequence (random). This method has to be overloaded by a model-dependant version when creating a **model navigator** class (cf. `ModelNavigator`). (Returns a **path**, i.e., a list of indexes. Generated sequence: cf. `Navigator.free_generation()`.)

Parameters

- **length** (*int*) – length of the generated sequence
- **new_max_continuity** (*int*) – new value for `self.max_continuity` (not changed if no parameter is given)
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **init** (*bool*) – reinitialise the navigation parameters ?, default : `False`. (True when starting a new generation)

- **equiv** (*function*) – Compararison function given as a lambda function, default: `self.equiv`.
- **print_info** (*bool*) – print the details of the navigation steps

Returns list of indexes of the generated path.

Return type list (*int*)

See also `Navigator.free_generation()`.

See also Example of overloaded method: `FactorOracleNavigator.free_navigation()`.

! **equiv** has to be consistent with the type of the elements in labels.

! The result **strongly depends** on the tuning of the parameters `self.max_continuity`, `self.avoid_repetitions_mode`, `self.no_empty_event`.

go_to_anterior_state_using_execution_trace (*index_in_navigation*)

This method is called when the run of a new query rewrites previously generated anticipations. It uses `self.execution_trace` to go back at the state where the navigator was at the “tiling time”.

Parameters `index_in_navigation` (*int*) – “tiling index” in the generated sequence

See also The list of the parameters of the model whose values are stored in the execution trace is defined in `self.execution_trace_parameters`.

is_taboo (*index*)

length_common_backward_context (*index_state1*, *index_state2*, *equiv=None*)

Length of the backward context shared by two states in the sequence.

Parameters **equiv** (*function*) – Compararison function given as a lambda function, default: `self.equiv`.

Returns Length of the longest equivalent sequences of labels before these states.

Return type *int*

! **equiv** has to be consistent with the type of the elements in labels.

length_common_forward_context (*index_state1*, *index_state2*, *equiv=None*)

Length of the forward context shared by two states in the sequence.

Parameters **equiv** (*function*) – Compararison function given as a lambda function, default: `self.equiv`.

Returns Length of the longest equivalent sequences of labels after these states.

Return type *int*

! **equiv** has to be consistent with the type of the elements in labels.

navigate_without_continuation (*authorized_indexes*)

Random state in the sequence if `self.no_empty_event` is True (else None).

Parameters **authorized_indexes** (*list(int)*) – list of authorized indexes to filter taboos, repetitions, and label when needed.

Returns index of the state

Return type *int*

record_execution_trace (*index_in_navigation*)

Stores in `self.execution_trace` the values of different parameters of the model when generating the event in the sequence at the index given in argument.

Parameters **index_in_navigation** (*int*) –

See also The list of the parameters of the model whose values are stored in the execution trace is defined in `self.execution_trace_parameters`.

reinit_navigation_param ()

(Re)initializes the navigation parameters (current navigation index, history of retrieved indexes, current continuity,...).

simply_guided_generation (*required_labels*, *new_max_continuity=None*, *forward_context_length_min=0*, *init=False*, *equiv=None*, *print_info=False*, *shift_index=0*, *break_when_none=False*)

Navigation in the sequence, simply guided step by step by an input sequence of label. Naive version of the method handling the navigation in a sequence when it is guided by target labels. This method has to be overloaded by a model-dependant version when creating a **model navigator** class (cf. `ModelNavigator`).

Parameters

- **required_labels** (*list*) – guiding sequence of labels
- **new_max_continuity** (*int*) – new value for `self.max_continuity` (not changed if no parameter is given)
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **init** (*bool*) – reinitialise the navigation parameters ?, default : False. (True when starting a new generation)
- **equiv** (*function*) – Comparison function given as a lambda function, default: `self.equiv`.
- **print_info** (*bool*) – print the details of the navigation steps

Returns generated sequence

Return type list

See also Example of overloaded method: `FactorOracleNavigator.free_navigation()`.

! **equiv** has to be consistent with the type of the elements in labels.

! The result **strongly depends** on the tuning of the parameters `self.max_continuity`, `self.avoid_repetitions_mode`, `self.no_empty_event`.

simply_guided_navigation (*required_labels*, *new_max_continuity=None*, *forward_context_length_min=0*, *init=False*, *equiv=None*, *print_info=False*, *shift_index=0*, *break_when_none=False*)

Navigation through the sequence, simply guided step by step by an input sequence of label. Naive version of the method handling the guided navigation in a sequence. This method has to be overloaded by a model-dependant version when creating a **model navigator** class (cf. `ModelNavigator`). (Returns a **path**, i.e., a list of indexes. Generated sequence: cf. `Navigator.simply_guided_generation()`.)

Parameters

- **required_labels** (*list*) – guiding sequence of labels
- **new_max_continuity** (*int*) – new value for `self.max_continuity` (not changed if no parameter is given)
- **forward_context_length_min** (*int*) – minimum length of the forward common context

- **init** (*bool*) – reinitialise the navigation parameters ?, default : False. (True when starting a new generation)
- **equiv** (*function*) – Comparison function given as a lambda function, default: self.equiv.
- **print_info** (*bool*) – print the details of the navigation steps

Returns list of indexes of the generated path.

Return type list (*int*)

See also `Navigator.simply_guided_generation()`.

See also Example of overloaded method: `FactorOracleNavigator.simply_guided_navigation()`.

! **equiv** has to be consistent with the type of the elements in labels.

! The result **strongly depends** on the tuning of the parameters `self.max_continuity`, `self.avoid_repetitions_mode`, `self.no_empty_event`.

update_history_and_taboos (*index_in_sequence*)

Increases the value associated to the index given in argument in `self.history_and_taboos`. Handles the taboos linked to `self.max_continuity`.

Parameters `index_in_sequence` (*int*) –

3.3 Meta Model Navigator

This module defines the main tool to create a new class of **model navigator**: the metaclass `MetaModelNavigator`. A model navigator is a class that implements different algorithms, strategies, and heuristics to navigate through a given model of sequence for analysis or creative applications, for example **generating new sequences using concatenative synthesis of events learned in the model**. The creation of the class `FactorOracleNavigator` in the file `ModelNavigator.py` can be considered as a tutorial for the metaclass `MetaModelNavigator`. When the whole library is loaded, the global dict `implemented_model_navigator_classes` automatically stores the different classes of model navigators available.

class `MetaModelNavigator.MetaModelNavigator` (*name, bases, dict_methods*)

Bases: `type`

`MetaModelNavigator` is a **metaclass**. A new class created using this metaclass is a **model navigator** and inherits from: 1) a class inheriting from `Model`, 2) a class inheriting from `Navigator`. The class `FactorOracleNavigator` introduced below is an example of model navigator created using this metaclass.

The definition of a new class of model navigator using this metaclass is quite easy: 1) chose two bases (a **model** inheriting from `Model`, a navigator inheriting from `Navigator`), 2) define the methods to overload `Navigator.simply_guided_navigation()` and `Navigator.free_navigation()`, 3) (optional) define other model-dependent slots and methods.

Example

```
>>> ##### Creation of a new class of model navigator
>>> # Define the 2 bases inheriting from the classes Model and Navigator,
    ↳ respectively
>>> tuple_bases = (MyModel, MyNavigator)
>>>
>>> # Define methods to overload Navigator.free_navigation() and Navigator.simply_
    ↳ guided_navigation()
```

```

>>> # (and other methods, including __init__ if needed)
>>> def free_navigation(my_model_navigator, length, new_max_continuity = None,
↳ forward_context_length_min = 0, init = False, equiv = None, print_info = False):
>>>     pass
>>>
>>> def simply_guided_navigation(factor_oracle_navigator, required_labels, new_
↳ max_continuity = None, forward_context_length_min = 0, init = False, equiv =
↳ None, print_info = False, shift_index = 0, break_when_none = False):
>>>     pass
>>>
>>> dict_methods = {"free_navigation" : free_navigation, "simply_guided_navigation
↳ " : simply_guided_navigation}
>>>
>>> # Creation of the new class of model navigator
>>> MyModelNavigator = MetaModelNavigator("MyModelNavigator", tuple_bases, dict_
↳ methods)

```

See also The creation of the class *FactorOracleNavigator* in the file *ModelNavigator.py* can be considered as a tutorial.

3.4 Factor Oracle Navigator

```

class ModelNavigator.FactorOracleNavigator (factor_oracle_navigator, sequence=[],
                                             labels=[], max_continuity=2, con-
                                             trol_parameters=[], history_parameters=[],
                                             equiv=<function <lambda>>)

```

Bases: *Model.FactorOracleNavigator.Navigator*

Factor Oracle Navigator class. This class implements heuristics of navigation through a Factor Oracle automaton for creative applications: different ways to find paths in the labels of the automaton to collect the associated contents and generate new sequences using concatenative synthesis. Original navigation heuristics, see Assayag, Bloch, “Navigating the Oracle: a heuristic approach”, in *Proceedings of the International Computer Music Conference 2007* (<https://hal.archives-ouvertes.fr/hal-01161388>).

See also Tutorial in *_Tutorials/FactorOracleNavigator_tutorial.py*.

See also This “model navigator” class is created with the metaclass *MetaModelNavigator*.

Example

```

>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels = [s[0] for s in sequence]
>>> FON = FactorOracleNavigator(sequence, labels)

```

```

filtered_continuations (factor_oracle_navigator, index_state, forward_context_length_min=0,
                        equiv=None)

```

Continuations from the state at index *index_state* in the automaton (see method *continuations*), and filtered continuations satisfying the constraints of taboos and repetitions (cf. *FactorOracleNavigator.history_and_taboos* and *FactorOracleNavigator.avoid_repetitions_mode*).

Parameters

- **index_state** (*int*) – start index
- **required_label** – label to read (optional)

- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **equiv** (*function*) – Comparison function given as a lambda function, default: `factor_oracle_navigator.equiv`.

Returns Indexes in the automaton of the possible continuations from the state at index `index_state` in the automaton.

Return type `tuple(list(int), list(int))`

See also `FactorOracleNavigator.continuations(...)`

! **equiv** has to be consistent with the type of the elements in labels.

filtered_continuations_with_label (*factor_oracle_navigator*, *index_state*, *required_label*, *forward_context_length_min=0*, *equiv=None*)

Continuations labeled by `required_label` from the state at index `index_state` in the automaton (see method `continuations`), and filtered continuations satisfying the constraints of taboos and repetitions (cf. `FactorOracleNavigator.history_and_taboos` and `FactorOracleNavigator.avoid_repetitions_mode`).

Parameters

- **index_state** (*int*) – start index
- **required_label** – label to read
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **equiv** (*function*) – Comparison function given as a lambda function, default: `factor_oracle_navigator.equiv`.

Returns Indexes in the automaton of the possible continuations from the state at index `index_state` in the automaton.

Return type `tuple(list(int), list(int))`

See also `FactorOracleNavigator.continuations_with_label(...)`

! **equiv** has to be consistent with the type of the elements in labels.

free_navigation (*factor_oracle_navigator*, *length*, *new_max_continuity=None*, *forward_context_length_min=0*, *init=False*, *equiv=None*, *print_info=False*)

Free navigation through the automaton. Returns a novel sequence being consistent with the internal logic of the sequence on which the automaton is built. (Returns a **path**, i.e., a list of indexes. Generated sequence: cf. `Navigator.free_generation()`.)

“Omax-like” navigation, see Assayag, Bloch, Chemillier, Cont, Dubnov “Omax brothers: A dynamic topology of agents for improvisation learning”, in *Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia* (<https://hal.archives-ouvertes.fr/hal-01161351>).

Parameters

- **length** (*int*) – length of the generated sequence
- **new_max_continuity** (*int*) – new value for `self.max_continuity` (not changed if no parameter is given)
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **init** (*bool*) – reinitialise the navigation parameters ?, default : `False`. (True when starting a new generation)

- **equiv** (*function*) – Compararison function given as a lambda function, default: `self.equiv`.
- **print_info** (*bool*) – print the details of the navigation steps

Returns list of indexes of the generated path.

Return type list (*int*)

See also `Navigator.free_generation()`

See also Tutorial in `FactorOracleNavigator_tutorial.py`.

! **equiv** has to be consistent with the type of the elements in labels.

! The result **strongly depends** on the tuning of the parameters `self.max_continuity`, `self.avoid_repetitions_mode`, `self.no_empty_event`.

Example

```
>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels = [s[0] for s in sequence]
>>> FON = FactorOracleNavigator(sequence, labels)
>>>
>>> FON.current_position_in_sequence = random.randint(0, FON.index_last_
↳ state())
>>> FON.avoid_repetitions_mode = 1
>>> FON.max_continuity = 2
>>> forward_context_length_min = 0
>>> generated_sequence = FON.free_generation(10, forward_context_length_min = _
↳ forward_context_length_min, print_info = True)
```

reinit_navigation_param (*factor_oracle_navigator*)

(Re)initializes the navigation parameters (current navigation index, history of retrieved indexes, current continuity,...).

simply_guided_navigation (*factor_oracle_navigator*, *required_labels*,
new_max_continuity=None, *forward_context_length_min=0*,
init=False, *equiv=None*, *print_info=False*, *shift_index=0*,
break_when_none=False)

Navigation through the automaton, simply guided step by step by an input sequence of label. Returns a novel sequence being consistent with the internal logic of the sequence on which the automaton is built, and matching the labels in `required_labels`. (Returns a **path**, i.e., a list of indexes. Generated sequence: cf. `Navigator.simply_guided_generation()`.)

Parameters

- **required_labels** (*list*) – guiding sequence of labels
- **new_max_continuity** (*int*) – new value for `self.max_continuity` (not changed if no parameter is given)
- **forward_context_length_min** (*int*) – minimum length of the forward common context
- **init** (*bool*) – reinitialise the navigation parameters ?, default : False. (True when starting a new generation)
- **equiv** (*function*) – Compararison function given as a lambda function, default: `self.equiv`.
- **print_info** (*bool*) – print the details of the navigation steps

Returns list of indexes of the generated path.

Return type list (int)

See also Navigator.simply_guided_generation()

See also Tutorial in FactorOracleNavigator_tutorial.py.

! equiv has to be consistent with the type of the elements in labels.

! The result **strongly depends on the tuning of the parameters self.max_continuity, self.avoid_repetitions_mode, self.no_empty_event.**

Example

```
>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels = [s[0] for s in sequence]
>>> FON = FactorOracleNavigator(sequence, labels)
>>>
>>> FON.current_position_in_sequence = random.randint(0, FON.index_last_
↳ state())
>>> FON.avoid_repetitions_mode = 1
>>> FON.max_continuity = 2
>>> FON.no_empty_event = True
>>> forward_context_length_min = 0
>>>
>>> guide = ['C', 'A', 'B', 'B', 'C', 'C', 'D']
>>> generated_sequence = FON.simply_guided_generation(guide, forward_context_
↳ length_min = forward_context_length_min, init = True, print_info = True)
```

HANDLE GENERATION

4.1 Query

The class *Query* and its subclasses define queries that are then processed by instances of class *Generator* or *GenerationHandler* to guide or constrain the generation of new sequences.

```
class Query.Query (start_date=0, start_unit='event', start_type='absolute', handle=[None],  
                  scope_duration=0, scope_unit='event', behaviour='merge')
```

A **query** guides or constrains the run of a generation model.

Parameters

- **handle** (*list*) – “Handle” of the query / sequence of required labels ([None] for free generation).
- **start** (*dict*) – Date concerned by the output of the query (details below).
- **start["date"]** (*int*) – Numerical value (unit and relative/absolute time: see below).
- **start["unit"]** (*str*) – Unit of the value given in start[“date”]: “event” or “ms”
- **start["type"]** (*str*) – Expressed in relative or absolute time ?: “relative” or “absolute”
- **scope** (*dict*) – Temporal horizon of the query (details below).
- **scope["duration"]** (*int*) – Numerical value (unit: see below).
- **scope["unit"]** (*str*) – Unit of the value given in scope[“duration”]: “event” or “ms”
- **behaviour** (*str*) – behaviour when a previous query concerned the same dates “merge” or “replace”
- **status** (*str*) – “waiting” or “being processed”

```
process (*args, **kwargs)
```

```
relative_to_absolute (current_performance_time_event=None, current_performance_time_ms=None)
```

```
Query.new_temporal_query_free_sequence_of_events (length=1, start_date=0,  
                                                  start_type='relative', behaviour='replace')
```

```
Query.new_temporal_query_sequence_of_events (handle=[], start_date=0,  
                                              start_type='relative', behaviour='replace')
```

4.2 Generator

This module defines agents generating new sequences from a “memory” (model of sequence) and generation queries. Main classes: *Generator* (oriented towards offline generation), *GenerationHandler* (oriented towards interactivity).

```
class Generator.GenerationHandler (sequence=[], labels=[], model_navigator='FactorOracleNavigator',
                                   equiv=<function                               <lambda>>,
                                   equiv_mod_interval=<function                 <lambda>>,
                                   authorized_transformations=[0],               se-
                                   quence_to_interval_fun=<function
                                   chord_labels_sequence_to_interval>,         continu-
                                   ity_with_future=[0.0, 1.0])
```

Bases: *Generator.Generator*

The class **GenerationHandler** introduces time management and planning for interactive applications and adds a pool of query, concurrency (e.g. processing of concurrent queries), the use of an execution trace etc. to the class *Generator*. More details in Nika, Bouche, Bresson, Chemillier, Assayag, “Guided improvisation as dynamic calls to an offline model”, in Proceedings of Sound and Music Computing conference 2015 (<https://hal.archives-ouvertes.fr/hal-01184642/document>), describing the first prototype of “improvisation handler”.

The key differences between *Generator* and *GenerationHandler* are:

- *Generator.receive_query()* / *GenerationHandler.receive_query()*
- *Generator.process_query()* / *GenerationHandler.process_query()*

Parameters

- **generation_trace** (*list*) – Whole output: current state of the sequence generated from the beginning (*GenerationHandler.start()*).
- **current_performance_time** (*dict*) – Current time of the performance, see below.
- **current_performance_time["event"]** (*int*) – Time expressed in events.
- **current_performance_time["ms"]** (*int*) – Time expressed in ms.
- **current_performance_time["last_update_event_in_ms"]** (*int*) – Date when the last timing information was received.
- **current_generation_time** (*dict*) – Current time of the generation, see below.
- **current_generation_time["event"]** (*int*) – Time expressed in events.
- **current_generation_time["ms"]** (*int*) – Time expressed in ms.
- **current_duration_event_ms** (*float*) – If all the events have (more or less) a same duration (e.g. a clock, pulsed music, non timed sequences...), this attribute is not None. It is then used to convert events into dates in ms.
- **query_pool_event** (list(*Query*)) – Pool of waiting queries expressed in events.
- **query_pool_ms** (list(*Query*)) – Pool of waiting queries expressed in ms.

See also *GeneratorBuilder*, automatic instantiation of Generator objects and GenerationHandler objects from annotation files.

See also Tutorial in `_Tutorials_/Generator_tutorial.py`.

Example

```

>>> labels = make_sequence_of_chord_labels(["d m7", "d m7", "g 7", "g 7", "c maj7",
↳ "c maj7", "c# maj7", "c# maj7", "d# m7", "d# m7", "g# 7", "g# 7", "c# maj7", "c
↳ # maj7"])
>>> sequence = make_sequence_of_chord_labels(["d m7(1)", "d m7(2)", "g 7(3)", "g
↳ 7(4)", "c maj7(5)", "c maj7(6)", "c# maj7(7)", "c# maj7(8)", "d# m7(9)", "d# m7(10)
↳ ", "g# 7(11)", "g# 7(12)", "c# maj7(13)", "c# maj7(14)"])
>>>
>>> print("\nCreation of a Generation Handler\nModel type = Factor
↳ Oracle\nSequence: {} \nLabels: {}".format(sequence, labels))
>>>
>>> authorized_intervals = range(-6,6)
>>> generation_handler = GenerationHandler(sequence = sequence, labels = labels,
↳ model_type = "FactorOracleNavigator", authorized_transformations = authorized_
↳ intervals, sequence_to_interval_fun = chord_labels_sequence_to_interval)
>>> generation_handler.memory.avoid_repetitions_mode = 1
>>> generation_handler.memory.max_continuity = 3
>>> generation_handler.memory.no_empty_event = False
>>> generation_handler.start()

```

estimation_date_event_of_ms (*date_ms*)

If all the events have (more or less) a same duration (e.g. a clock, pulsed music, non timed sequences...), `self.current_duration_event_ms` is not None. It is then used to convert dates in ms into indexes of events.

Parameters `date_ms` (*int*) – date in ms to convert

Returns estimated corresponding index of event (or None)

Return type `int`

estimation_date_ms_of_event (*num_event*)

If all the events have (more or less) a same duration (e.g. a clock, pulsed music, non timed sequences...), `self.current_duration_event_ms` is not None. It is then used to convert indexes of events into dates in ms.

Parameters `num_event` (*int*) – index of event to convert

Returns estimated corresponding date in ms (or None)

Return type `int`

inc_performance_time (*inc_event=None, inc_ms=None*)

index_previously_generated_event_date_ms (*date_query*)

process_prioritary_query (*unit=None, print_info=False*)

Processes the prioritary query in the query pool.

process_query (*query, print_info=False*)

The key differences between *Generator* and *GenerationHandler* are:

- *Generator.receive_query()* / *GenerationHandler.receive_query()*
- *Generator.process_query()* / *GenerationHandler.process_query()*

This methods takes time into account: in addition to what *Generator.process_query()* does, it compares the attribute `Query.start` of the query and `self.current_performance_time` to call `Navigator.go_to_anterior_state_using_execution_trace()` if needed. This way, it ensures consistency at tiling time when rewriting previously generated anticipations. As in *Generator.process_query()* the output of this query is stored in `self.current_generation_output`. In addition it is inserted at the right place in the whole output history `self.generation_trace`.

Parameters `query` (*Query*) –

Returns `query.start["date"]` (converted to “absolute” value)

Return type `int`

receive_query (*query*, *print_info=False*)

The key differences between *Generator* and *GenerationHandler* are:

- *Generator.receive_query()* / *GenerationHandler.receive_query()*
- *Generator.process_query()* / *GenerationHandler.process_query()*

Inserts the received query in the query pool. Handles the interaction between (running and/or waiting) queries: merging compatible queries, killing outdated queries... The queries in the query pool are then run in due time. **TODO: for the moment only “append” and immediate processing.**

Parameters `query` (*Query*) –

Example

```
>>> labels = make_sequence_of_chord_labels(["d m7", "d m7", "g 7", "g 7", "c#
↪maj7", "c maj7", "c# maj7", "c# maj7", "d# m7", "d# m7", "g# 7", "g# 7", "c#
↪maj7", "c# maj7"])
>>> sequence = make_sequence_of_chord_labels(["d m7(1)", "d m7(2)", "g 7(3)",
↪g 7(4)", "c maj7(5)", "c maj7(6)", "c# maj7(7)", "c# maj7(8)", "d# m7(9)", "d
↪# m7(10)", "g# 7(11)", "g# 7(12)", "c# maj7(13)", "c# maj7(14)"])
>>>
>>> print("\nCreation of a Generation Handler\nModel type = Factor_
↪Oracle\nSequence: {}\nLabels: {}".format(sequence, labels))
>>>
>>> authorized_intervals = range(-6,6)
>>> generation_handler = GenerationHandler(sequence = sequence, labels =
↪labels, model_type = "FactorOracleNavigator", authorized_tranformations =
↪authorized_intervals, sequence_to_interval_fun = chord_labels_sequence_to_
↪interval)
>>> generation_handler.memory.avoid_repetitions_mode = 1
>>> generation_handler.memory.max_continuity = 3
>>> generation_handler.memory.no_empty_event = False
>>> generation_handler.start()
>>>
>>> scenario = make_sequence_of_chord_labels(["g m7", "g m7", "c 7", "c 7",
↪f maj7", "f maj7"])
>>> query= new_temporal_query_sequence_of_events(scenario)
>>> print("\n/!\ Receiving and processing a new query: /!\ \n{)".
↪format(query))
>>> generation_handler.receive_query(query = query, print_info = False)
>>> print("Output of the run: {}".format(generation_handler.current_
↪generation_output))
>>> print("/!\ Updated buffered improvisation: {} /!\ ".format(generation_
↪handler.generation_trace))
>>>
>>> query= new_temporal_query_free_sequence_of_events(length = 3, start_date_
↪= 4, start_type = "absolute")
>>> print("\n/!\ Receiving and processing a new query: /!\ \n{)".
↪format(query))
>>> generation_handler.receive_query(query = query, print_info = False)
>>> print("Output of the run: {}".format(generation_handler.current_
↪generation_output))
>>> print("/!\ Updated buffered improvisation: {} /!\ ".format(generation_
↪handler.generation_trace))
```

```

start ()
    Sets self.current_performance_time to 0.

update_performance_time (date_event=None, date_ms=None)

class Generator.Generator (sequence=[], labels=[], model_navigator='FactorOracleNavigator',
                           equiv=<function <lambda>>, equiv_mod_interval=<function
                           <lambda>>, authorized_transformations=[0], se-
                           quence_to_interval_fun=<function chord_labels_sequence_to_interval>,
                           continuity_with_future=[0.0, 1.0])

```

Bases: `object`

The class **Generator** embeds a **model navigator** as “memory” (cf. metaclass *MetaModelNavigator*) and processes **queries** (class *Query*) to generate new sequences. This class uses pattern matching techniques (cf. *PrefixIndexing*) to enrich the navigation and generation methods offered by the chosen model with (“ImproteK-like”) anticipative behaviour. More information on “scenario-based generation”: see Nika, “Guiding Human-Computer Music Improvisation: introducing Authoring and Control with Temporal Scenarios”, PhD Thesis, UPMC Paris 6, Ircam, 2016 (<https://tel.archives-ouvertes.fr/tel-01361835>) and Nika, Chemillier, Assayag, “ImproteK: introducing scenarios into human-computer music improvisation”, ACM Computers in Entertainment, Special issue on Musical metacreation Part I, 2017 (<https://hal.archives-ouvertes.fr/hal-01380163>).

The key differences between *Generator* and *GenerationHandler* are:

- *Generator.receive_query()* / *GenerationHandler.receive_query()*
- *Generator.process_query()* / *GenerationHandler.process_query()*

Parameters

- **model_navigator** (*str*) –
- **memory** (cf. *ModelNavigator* and *MetaModelNavigator*) – “Model navigator” inheriting from (a subclass of) *Model* and (a subclass of) *Navigator*.
- **initial_query** (*bool*) –
- **current_generation_query** (*Query*) –
- **current_generation_output** (*list*) –
- **transfo_current_generation_output** (*list*) –
- **continuity_with_future** (*list*) –
- **current_transformation_memory** (cf. *Transforms*) –
- **authorized_transformations** (*list(int)*) –
- **sequence_to_interval_fun** (*function*) –
- **equiv_mod_interval** (*function*) –

See also *GeneratorBuilder*, automatic instantiation of Generator objects and GenerationHandler objects from annotation files.

See also Tutorial in `_Tutorials_/Generator_tutorial.py`.

Example

```

>>> sequence_1 = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels_1 = [s[0] for s in sequence_1]
>>> generator_1 = Generator(sequence=sequence_1, labels=labels_1, model_navigator_
↳ "FactorOracleNavigator")
>>>
>>> sequence_2 = make_sequence_of_chord_labels(["d m7(1)", "d m7(2)", "g 7(3)",
↳ "g 7(4)", "c maj7(5)", "c maj7(6)", "c# maj7(7)", "c# maj7(8)", "d# m7(9)", "d#
↳ m7(10)", "g# 7(11)", "g# 7(12)", "c# maj7(13)", "c# maj7(14)"])
>>> labels_2 = make_sequence_of_chord_labels(["d m7", "d m7", "g 7", "g 7", "c
↳ maj7", "c maj7", "c# maj7", "c# maj7", "d# m7", "d# m7", "g# 7", "g# 7", "c# maj7",
↳ "c# maj7"])
>>> authorized_intervals = range(-2, 6)
>>> generator_2 = Generator(sequence = sequence_2, labels = labels_2, model_
↳ navigator = "FactorOracleNavigator", authorized_transformations = authorized_
↳ intervals, sequence_to_interval_fun = chord_labels_sequence_to_interval)

```

decode_memory_with_current_transfo()

decode_memory_with_transfo (*transform*)

Apply the reciprocal transformation of the transformation given in argument to self.memory.sequence and self.memory.label.

Parameters **transform** (cf. *Transforms*) –

encode_memory_with_current_transfo()

encode_memory_with_transfo (*transform*)

Apply the transformation given in argument to self.memory.sequence and :attr:`self.memory.label`.

Parameters **transform** (cf. *Transforms*) –

filter_using_history_and_taboos (*list_of_indexes*)

formatted_output_couple_content_transfo()

generation_matching_query (*query*, *print_info=False*)

Launches the run of a generation process corresponding to **self.current_generation_query** (more precisely

- *Generator.handle_free_generation()*, or
- *Generator.handle_generation_matching_label()*, or
- *Generator.handle_scenario_based_generation()*.

The generated sequence is stored in **self.current_generation_output**.

Parameters **query** (*Query*) –

handle_free_generation (*length*, *print_info=False*)

Generates a sequence using the method *free_generation()* of the model navigator (cf. *ModelNavigator*) in **self.memory**. *Generator.encode_memory_with_current_transfo()* and *Generator.decode_memory_with_current_transfo()* are respectively called before and after this generation.

Parameters **length** (*int*) – required length of the sequence

Returns generated sequence

Return type list

See also `free_generation()`

See also `MetaModelNavigator`

handle_generation_matching_label (*label*, *print_info=False*)

Generates a single event using the method `simply_guided_generation()` of the model navigator (cf. `ModelNavigator`) in `self.memory`. `Generator.encode_memory_with_current_transfo()` and `Generator.decode_memory_with_current_transfo()` are respectively called before and after this generation.

Parameters `label` (type of the elements in `self.memory.label`) – required label

Returns generated event

Return type type of the elements in `self.memory.sequence`

See also `simply_generation()`

See also `MetaModelNavigator`

handle_scenario_based_generation (*list_of_labels*, *print_info=False*)

Generates a sequence matching a “scenario” (a list of labels). The generation process takes advantage of the scenario to introduce anticipatory behaviour, that is, continuity with the future of the scenario. The generation process is divided in successive “generation phases”, cf. `handle_scenario_based_generation_one_phase()`.

Parameters `list_of_labels` (*list* or *str*) – “scenario”, required labels

Returns generated sequence

Return type list

handle_scenario_based_generation_one_phase (*list_of_labels*, *print_info=False*, *shift_index=0*)

Parameters `list_of_labels` (*list* or *str*) – “current scenario”, suffix of the scenario given in argument to `Generator.handle_scenario_based_generation()`.

A “scenario-based” generation phase:

1. Anticipation step: looking for an event in the memory sharing a common future with the current scenario.
2. Navigation step: starting from the starting point found in the first step, navigation in the memory using `simply_guided_generation()` until launching a new phase is necessary.

learn_event (*state*, *label*)

Learn a new event in the memory (model navigator).

learn_sequence (*new_sequence*)

Learn a new sequence in the memory (model navigator).

process_query (*query*, *print_info=False*)

The key differences between `Generator` and `GenerationHandler` are:

- `Generator.receive_query()` / `GenerationHandler.receive_query()`
- `Generator.process_query()` / `GenerationHandler.process_query()`

This methods stores the query given in argument in `self.current_generation_query` and calls `Generator.generation_matching_query()` to run the execution of a generation process adapted to `Query.handle` and `Query.scope`.

Parameters **query** (*Query*) –

receive_query (*query*, *print_info=False*)

The key differences between *Generator* and *GenerationHandler* are:

- *Generator.receive_query()* / *GenerationHandler.receive_query()*
- *Generator.process_query()* / *GenerationHandler.process_query()*

Here, a query is processed as soon as it is received.

Parameters **query** (*Query*) –

See also *Generator.process_query()*

Example

```
>>> sequence = ['A1', 'B1', 'B2', 'C1', 'A2', 'B3', 'C2', 'D1', 'A3', 'B4', 'C3']
>>> labels = [s[0] for s in sequence]
>>> generator = Generator(sequence=sequence, labels=labels, model_navigator =
↳ "FactorOracleNavigator")
>>> print("\nProcessing query 1 - generation guided by a scenario:")
>>> query_1 = new_temporal_query_sequence_of_events(['C', 'A', 'B', 'B', 'C', 'C',
↳ 'D'])
>>> generator.receive_query(query = query_1, print_info = True)
>>> print("Output: {}".format(generator.current_generation_output))
>>>
>>> print("\nAfter this generation phase:")
>>> print("History and taboos: {}".format(generator.memory.history_and_
↳ taboos))
>>> print("Current navigation index: {}".format(generator.memory.current_
↳ position_in_sequence))
>>>
>>> print("\nProcessing query 2 - free:")
>>> query_2 = new_temporal_query_free_sequence_of_events(length = 4)
>>> generator.receive_query(query = query_2, print_info = True)
>>> print("Output: {}".format(generator.current_generation_output))
>>>
>>> print("\nAfter this generation phase:")
>>> print("History and taboos: {}".format(generator.memory.history_and_
↳ taboos))
```

PATTERN MATCHING ALGORITHMS

5.1 Prefix indexing algorithms

Algorithms introduced in Nika, “Guiding Human-Computer Music Improvisation: introducing Authoring and Control with Temporal Scenarios”, PhD Thesis, UPMC Paris 6, Ircam, 2016 (<https://tel.archives-ouvertes.fr/tel-01361835>) and Nika, Chemillier, Assayag, “ImproteK: introducing scenarios into human-computer music improvisation”, ACM Computers in Entertainment, Special issue on Musical metacreation Part I, 2017 (<https://hal.archives-ouvertes.fr/hal-01380163>).

Tutorial in `_Tutorials_/PrefixIndexing_tutorial.py`.

`PrefixIndexing.add_shorter_prefixes` (*longest_prefixes_pattern_left_pos_in_sequence*, *internal_prefixes_in_pattern*, *j*, *i*, *print_info=0*)

Sub-routine used in `PrefixIndexing.prefix_indexing()`.

Given the prefixes of the pattern found **so far** in the sequence at step *i,j* in `PrefixIndexing.prefix_indexing()` (only the longest in the region [*j-length_longest_prefix*(step *i,j*),*j*-1]), return **all the prefixes** of the pattern in the sequence at step *i,j* using the **previously computed internal prefixes in the pattern** (`PrefixIndexing.failure_function_and_right_pos_prefixes()`).

Parameters

- **longest_prefixes_pattern_left_pos_in_sequence** (*dict* (*int* -> *list* of *int*)) – prefixes of the pattern found **so far** (only the longest in region [*j-length_longest_prefix*,*j*-1]) in the sequence at step *i,j* of the research in `PrefixIndexing.prefix_indexing` (key = length, value = list of left positions of the prefixes of the pattern of length ‘length’ in the sequence)
- **internal_prefixes_in_pattern** (*dict* (*int* -> *list* of *int*)) – prefixes of the pattern in itself **resulting from a call to** `PrefixIndexing.failure_function_and_right_pos_prefixes()` (key = index in the pattern (from 0), value = list: lengths of the prefixes of the pattern in itself ending at the corresponding index)
- **j** (*int*) – current position of the cursor in the sequence
- **i** (*int*) – current position of the cursor in the pattern
- **print_info** (*int*) – print the details of the research?

Returns all the prefixes of the pattern of the sequence at step *i,j* of the research (`PrefixIndexing.prefix_indexing()`) (key = length, value = list: left positions of prefixes of length ‘length’)

Return type `dict` (*int* -> *list* of *int*)

`PrefixIndexing.failure_function` (*pattern*, *equiv*=<function <lambda>>)

Failure function from the Morris & Pratt algorithm.

Parameters

- **pattern** (*list* or *str*) – pattern on which the failure function is built
- **equiv** (*function*) – comparison function given as a lambda function, default: ==

Returns failure function (key = index in the pattern (from 0), value = failure_function[index])

Return type `dict` (int -> int)

Seealso **Tutorial in** `_Tutorials_/Generator_tutorial.py`

! **equiv** has to be consistent with the type of the elements in labels.

Example

```
>>> failure_function([1,2,3,2,1,2,3,1,2,3], equiv = (lambda x,y : x%2 == y%2))
```

`PrefixIndexing.failure_function_and_right_pos_prefixes` (*pattern*, *equiv*=<function
<lambda>>)

Failure function built on the pattern, and right positions of the prefixes of the pattern in itself.

Parameters

- **pattern** (*list* or *str*) – pattern on which the failure function is built
- **equiv** (*function*) – comparison function given as a lambda function, default: ==

Returns Failure function built on the pattern (key = index in the pattern (from 0), value = failure_function[index]), and right positions of the prefixes of the pattern in itself (key = index in the pattern (from 0), value = list: lengths of the prefixes of the pattern in itself ending at the corresponding index).

Return type `tuple` (`dict`(int -> int), `dict`(int -> list of int))

Seealso **Tutorial in** `_Tutorials_/Generator_tutorial.py`

! **equiv** has to be consistent with the type of the elements in labels.

Example

```
>>> pattern = [1,2,3,2,1,2,3,1,2,3]
>>> failure_dict, lengths_ending_prefixes_dict = failure_function_and_right_pos_
↳ prefixes(pattern)
>>> for idx,length in lengths_ending_prefixes_dict.items():
>>>     print("Index {}: right position of prefix(es) of {} in itself of_
↳ length(s): {}".format(idx, pattern, length))
```

`PrefixIndexing.filtered_prefix_indexing` (*sequence*, *pattern*, ***args*)

Filtered index of the prefixes of a pattern in a sequence (filtered regarding lengths and positions).

Parameters

- **sequence** (*list* or *str*) –
- **pattern** (*list* or *str*) –
- **authorized_indexes** (*list* (*int*)) – [args] list of authorized indexes to filter the results
- **length_interval** (*tuple* (int, int): absolute lengths** of the prefixes **or** *tuple* (float, float): fractions of the length of the longest prefix before filtering) – [args] interval of length to filter the results.

- **equiv** (*function*) – [args] compararison function given as a lambda function, default: `==`
- **print_info** (*int*) – [args] print the details of the research?

Returns prefixes of the pattern in the sequence after filtering (key = length, value = list of left positions of prefixes of the pattern of length 'length' in the sequence) **and** length of the longest prefix

Return type `tuple (dict (int -> list), int)`

Seealso Tutorial in `PrefixIndexing_tutorial.py`

! equiv has to be consistent with the type of the elements in labels.

Example

```
>>> pattern = [1,2,3,1,2,4]
>>> sequence = [1,2,3,1,2,1,1,2,3]
>>> authorized_indexes = [1,2,3,7,8]
>>> length_interval = 1,3
>>>
>>> prefixes, length_longest_prefix = filtered_prefix_indexing(sequence, pattern,
↳ authorized_indexes = authorized_indexes, length_interval = length_interval,
↳ equiv = (lambda x,y : x%2 == y%2), print_info = 0)
>>> print("\nPrefixes of \n{} \nin \n{}\nusing the user-defined comparison_
↳ function " == %2"\nAuthorized indexes = {} - Authorized length interval_
↳ (absolute length) = {}:\n----".format(pattern, sequence, authorized_indexes,
↳ length_interval))
>>>     for length,list_of_left_pos_in_sequence in prefixes.items():
>>>         print("Length {}: at left position(s) {}".format(length,list_of_left_pos_
↳ in_sequence))
```

```
>>> pattern = [1,2,3,1,2,4]
>>> sequence = [1,2,3,1,2,1,1,2,3]
>>> length_interval = 1.0/2, 4.0/5
>>>
>>> prefixes, length_longest_prefix = filtered_prefix_indexing(sequence, pattern,
↳ length_interval = length_interval, equiv = (lambda x,y : x%2 == y%2), print_
↳ info = 0)
>>> print("\nPrefixes of \n{} \nin \n{}\nusing the user-defined comparison_
↳ function " == %2"\nAuthorized length interval (fraction of maximum length_
↳ before filtering) = {}:\n----".format(pattern, sequence, length_interval))
>>> for length,list_of_left_pos_in_sequence in prefixes.items():
>>>     print("Length {}: at left position(s) {}".format(length,list_of_left_pos_
↳ in_sequence))
```

`PrefixIndexing.prefix_indexing` (*sequence*, *pattern*, ***args*)

Index the prefixes of a pattern in a sequence.

Parameters

- **sequence** (*list* or *str*) –
- **pattern** (*list* or *str*) –
- **equiv** (*function*) – [args] compararison function given as a lambda function, default: `==`
- **print_info** (*int*) – [args] print the details of the research?

Returns prefixes of the pattern in the sequence (key = length, value = list of left positions of prefixes of the pattern of length 'length' in the sequence) **and** length of the longest prefix

Return type tuple (dict (int -> list), int)

Seealso Tutorial in PrefixIndexing_tutorial.py

! equiv has to be consistent with the type of the elements in labels.

Example

```
>>> pattern = "ABCD"
>>> sequence = "ABCDABAABC"
>>>
>>> prefixes, length_longest_prefix = prefix_indexing(sequence, pattern)
>>> print("\nPrefixes of \n{} \nin \n{}:".format(pattern, sequence))
>>> for length, list_of_left_pos_in_sequence in prefixes.items():
>>>     print("Prefixes of length {}: at left position(s) {}".format(length, list_
↳ of_left_pos_in_sequence))
```

```
>>> pattern = [{"one", "1"}, {"three", "THREE"}, {"two", "two"}, {"two", "TWO"}, [
↳ "four", "FOUR"], [{"three", "THREE"}, \
>>> [{"three", "3"}, {"one", "ONE"}]
>>> sequence = [{"two", "two"}, {"one", "1"}, {"three", "3"}, {"two", "two"}, {"two",
↳ "TWO"}, {"one", "1"}, {"three", "three"}, \
>>> [{"two", "TWO"}, {"two", "TWO"}, {"four", "FOUR"}, {"four", "FOUR"}, {"one", "1"}, [
↳ "three", "THREE"}, {"two", "two"}, {"three", "THREE"}, \
>>> [{"three", "3"}, {"one", "ONE"}]
>>>
>>> prefixes, length_longest_prefix = prefix_indexing(sequence, pattern, equiv =
↳ (lambda x, y: x[0] == y[0]))
>>> print("\nPrefixes of \n{} \nin \n{}\nusing the user-defined comparison_
↳ function '1st elem. == 1st elem.':".format(pattern, sequence))
>>> for length, list_of_left_pos_in_sequence in prefixes.items():
>>>     print("Length {}: at left position(s) {}".format(length, list_of_left_pos_
↳ in_sequence))
```

5.2 Using Intervals

Some subclasses of `Label` enable to define a notion of interval. The tools defined in this module handle such possibilities. **Tutorial in** `_Tutorials/Label_and_intervals_tutorial.py`.

!... documentation in progress... !

EVENT, LABEL, TRANSFORMATION

6.1 Label

Definition of alphabets of labels to build sequences and use them in creative applications.

/!... documentation in progress... /!

6.2 Transforms

Class defining transformations on labels and contents.

/!... documentation in progress... /!

OSC COMMUNICATION

7.1 OSC AGENT

Class defining an OSC server embedding an instance of class *GenerationHandler*. See the different tutorials accompanied with Max patches.

/!... documentation in progress... /!

MEMORY AND CORPUS

8.1 Generator Builder

Tools and functions to create instances of classes *Generator* and *GenerationHandler* from a json file / dict defining a sequence of events with metadata.. Example of the required json / dict format: `_Tutorial_/ExamplesCorpus/ExampleDictMemory.json`.

```
GeneratorBuilder.extract_labels_and_contents_from_dict_memory(dict_memory,  
                                                             keys_labels,  
                                                             keys_contents)
```

Extracts a sequence of labels and sequence of contents from a dict / json file defining a sequence of events with metadata..

Parameters

- **dict_memory** (*dict*) – sequence of events with metadata
- **keys_labels** (*str*) – key of the field in the dict / json file that will be considered as label.
- **keys_contents** (*str*) – key of the field in the dict / json file that will be considered as content.

Returns sequence of labels, sequence of contents

Return type list, list

See also Example of the required json / dict format: `_Tutorial_/ExamplesCorpus/ExampleDictMemory.json`.

The dict / json file always contain fields that can be used with the following keys (for the labels as well as the contents):

- “absolute_time”: start date of the event, absolute time
- “relative_time”: start date of the event, relative time
- “tempo”: local tempo
- “index”: index in the sequence

```
GeneratorBuilder.new_generation_handler_from_dict_memory(dict_memory,
                                                         keys_labels,
                                                         keys_contents,
                                                         model_navigator='FactorOracleNavigator',
                                                         equiv=<function
                                                         <lambda>>,
                                                         equiv_mod_interval=<function
                                                         <lambda>>,      autho-
                                                         rized_transformations=[0],
                                                         se-
                                                         quence_to_interval_fun=<function
                                                         chord_labels_sequence_to_interval>,
                                                         continu-
                                                         ity_with_future=[0.0,
                                                         1.0])
```

Creates an instance of class *GenerationHandler* from a dict defining a sequence of events with metadata..

Parameters

- **dict_memory** (*dict*) – sequence of events with metadata
- **keys_labels** (*str*) – key of the field in the dict / json file that will be considered as label.
- **keys_contents** (*str*) – key of the field in the dict / json file that will be considered as content.

Other parameters: see *GenerationHandler*.

Returns Generation handler

Return type *GenerationHandler*

See also Example of the required json / dict format: `_Tutorial_/ExamplesCorpus/ExampleDictMemory.json`.

See also **Tutorial in** `_Tutorials_/GeneratorBuilder_tutorial.py`.

The dict / json file always contain fields that can be used with the following keys (for the labels as well as the contents):

- “absolute_time”: start date of the event, absolute time
- “relative_time”: start date of the event, relative time
- “tempo”: local tempo
- “index”: index in the sequence

```

GeneratorBuilder.new_generation_handler_from_json_file (path_json_file,
                                                    keys_labels, keys_contents,
                                                    model_navigator='FactorOracleNavigator',
                                                    equiv=<function
<lambda>>,
                                                    equiv_mod_interval=<function
<lambda>>,          autho-
rized_transformations=[0],
                                                    se-
quence_to_interval_fun=<function
chord_labels_sequence_to_interval>,
                                                    continu-
ity_with_future=[0.0,
1.0])

```

Creates an instance of class *GenerationHandler* from a json file defining a sequence of events with meta-data..

Parameters

- **path_json_file** (*str*) – path of the json file defining a sequence of events with meta-data.
- **keys_labels** (*str*) – key of the field in the dict / json file that will be considered as label.
- **keys_contents** (*str*) – key of the field in the dict / json file that will be considered as content.

Other parameters: see *GenerationHandler*.

Returns Generation handler

Return type *GenerationHandler*

See also Example of the required json / dict format: `_Tutorial_/ExamplesCorpus/ExampleDictMemory.json`.

See also Tutorial in `_Tutorials_/GeneratorBuilder_tutorial.py`.

The dict / json file always contain fields that can be used with the following keys (for the labels as well as the contents):

- “absolute_time”: start date of the event, absolute time
- “relative_time”: start date of the event, relative time
- “tempo”: local tempo
- “index”: index in the sequence

```

GeneratorBuilder.new_generator_from_dict_memory (dict_memory,          keys_labels,
                                                    keys_contents,
                                                    model_navigator='FactorOracleNavigator',
                                                    equiv=<function          <lambda>>,
                                                    equiv_mod_interval=<function
<lambda>>,          autho-
rized_transformations=[0],          se-
quence_to_interval_fun=<function
chord_labels_sequence_to_interval>,
                                                    continuity_with_future=[0.0, 1.0])

```

Creates an instance of class *Generator* from a dict defining a sequence of events with metadata..

Parameters

- **dict_memory** (*dict*) – sequence of events with metadata
- **keys_labels** (*str*) – key of the field in the dict / json file that will be considered as label.
- **keys_contents** (*str*) – key of the field in the dict / json file that will be considered as content.

Other parameters: see [Generator](#).

Returns Generator

Return type [Generator](#)

See also Example of the required json / dict format: `_Tutorial_/ExamplesCorpus/ExampleDictMemory.json`.

See also **Tutorial in** `_Tutorials_/GeneratorBuilder_tutorial.py`.

The dict / json file always contain fields that can be used with the following keys (for the labels as well as the contents):

- “absolute_time”: start date of the event, absolute time
- “relative_time”: start date of the event, relative time
- “tempo”: local tempo
- “index”: index in the sequence

```
GeneratorBuilder.new_generator_from_json_file(path_json_file,
                                             keys_labels,
                                             keys_contents,
                                             model_navigator='FactorOracleNavigator',
                                             equiv=<function          <lambda>>,
                                             equiv_mod_interval=<function
                                             <lambda>>,
                                             autho-
                                             rized_transformations=[0],
                                             se-
                                             quence_to_interval_fun=<function
                                             chord_labels_sequence_to_interval>,
                                             continuity_with_future=[0.0, 1.0])
```

Creates an instance of class [Generator](#) from a json file defining a sequence of events with metadata..

Parameters

- **path_json_file** (*str*) – path of the json file defining a sequence of events with meta-data.
- **keys_labels** (*str*) – key of the field in the dict / json file that will be considered as label.
- **keys_contents** (*str*) – key of the field in the dict / json file that will be considered as content.

Other parameters: see [Generator](#).

Returns Generator

Return type [Generator](#)

See also Example of the required json / dict format: `_Tutorial_/ExamplesCorpus/ExampleDictMemory.json`.

See also **Tutorial in** `_Tutorials_/GeneratorBuilder_tutorial.py`.

The dict / json file always contain fields that can be used with the following keys (for the labels as well as the contents):

- “absolute_time”: start date of the event, absolute time
- “relative_time”: start date of the event, relative time
- “tempo”: local tempo
- “index”: index in the sequence

/!... documentation in progress... /!

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

g

Generator, [23](#)

GeneratorBuilder, [39](#)

i

Intervals, [34](#)

l

Label, [35](#)

m

MetaModelNavigator, [18](#)

Model, [7](#)

n

Navigator, [13](#)

o

OSCAgent, [37](#)

p

PrefixIndexing, [31](#)

q

Query, [23](#)

t

Transforms, [35](#)

A

add_direct_transition() (Model.FactorOracle method), 8
 add_factor_link() (Model.FactorOracle method), 8
 add_shorter_prefixes() (in module PrefixIndexing), 31
 add_suffix_link() (Model.FactorOracle method), 8
 authorize_indexes() (Navigator.Navigator method), 14

B

build() (Model.Model method), 12

C

continuations() (Model.FactorOracle method), 8
 continuations_with_jump() (Model.FactorOracle method), 8
 continuations_with_label() (Model.FactorOracle method), 9

D

decode_memory_with_current_transfo() (Generator.Generator method), 28
 decode_memory_with_transfo() (Generator.Generator method), 28
 delete_taboos() (Navigator.Navigator method), 14

E

encode_memory_with_current_transfo() (Generator.Generator method), 28
 encode_memory_with_transfo() (Generator.Generator method), 28
 estimation_date_event_of_ms() (Generator.GenerationHandler method), 25
 estimation_date_ms_of_event() (Generator.GenerationHandler method), 25
 extract_labels_and_contents_from_dict_memory() (in module GeneratorBuilder), 39

F

FactorOracle (class in Model), 7
 FactorOracleNavigator (class in ModelNavigator), 19
 failure_function() (in module PrefixIndexing), 31
 failure_function_and_right_pos_prefixes() (in module PrefixIndexing), 32

filter_using_history_and_taboos() (Generator.Generator method), 28
 filter_using_history_and_taboos() (Navigator.Navigator method), 14
 filtered_continuations() (ModelNavigator.FactorOracleNavigator method), 19
 filtered_continuations_with_label() (ModelNavigator.FactorOracleNavigator method), 20
 filtered_prefix_indexing() (in module PrefixIndexing), 32
 find_matching_label_without_continuation() (Navigator.Navigator method), 14
 follow_continuation_using_transition() (Model.FactorOracle method), 9
 follow_continuation_with_jump() (Model.FactorOracle method), 9
 follow_reverse_suffix_links_from() (Model.FactorOracle method), 10
 follow_suffix_links_from() (Model.FactorOracle method), 10
 follow_suffix_links_then_reverse_from() (Model.FactorOracle method), 10
 forbid_indexes() (Navigator.Navigator method), 15
 formatted_output_couple_content_transfo() (Generator.Generator method), 28
 free_generation() (Navigator.Navigator method), 15
 free_navigation() (ModelNavigator.FactorOracleNavigator method), 20
 free_navigation() (Navigator.Navigator method), 15
 from_state_read_label() (Model.FactorOracle method), 10

G

generation_matching_query() (Generator.Generator method), 28
 GenerationHandler (class in Generator), 24
 Generator (class in Generator), 27
 Generator (module), 23
 GeneratorBuilder (module), 39
 go_to_anterior_state_using_execution_trace() (Navigator.Navigator method), 16

H

`handle_free_generation()` (Generator.Generator method), 28
`handle_generation_matching_label()` (Generator.Generator method), 29
`handle_scenario_based_generation()` (Generator.Generator method), 29
`handle_scenario_based_generation_one_phase()` (Generator.Generator method), 29

I

`inc_performance_time()` (Generator.GenerationHandler method), 25
`index_last_state()` (Model.Model method), 13
`index_previously_generated_event_date_ms()` (Generator.GenerationHandler method), 25
`init_model()` (Model.FactorOracle method), 10
`init_model()` (Model.Model method), 13
Intervals (module), 34
`is_recognized()` (Model.FactorOracle method), 10
`is_taboo()` (Navigator.Navigator method), 16

L

Label (module), 35
`learn_event()` (Generator.Generator method), 29
`learn_event()` (Model.FactorOracle method), 11
`learn_event()` (Model.Model method), 13
`learn_sequence()` (Generator.Generator method), 29
`learn_sequence()` (Model.Model method), 13
`length_common_backward_context()` (Navigator.Navigator method), 16
`length_common_forward_context()` (Navigator.Navigator method), 16

M

MetaModelNavigator (class in MetaModelNavigator), 18
MetaModelNavigator (module), 18
Model (class in Model), 12
Model (module), 7

N

`navigate_without_continuation()` (Navigator.Navigator method), 16
Navigator (class in Navigator), 13
Navigator (module), 13
`new_generation_handler_from_dict_memory()` (in module GeneratorBuilder), 39
`new_generation_handler_from_json_file()` (in module GeneratorBuilder), 40
`new_generator_from_dict_memory()` (in module GeneratorBuilder), 41
`new_generator_from_json_file()` (in module GeneratorBuilder), 42

`new_temporal_query_free_sequence_of_events()` (in module Query), 23
`new_temporal_query_sequence_of_events()` (in module Query), 23

O

OSCAgent (module), 37

P

`prefix_indexing()` (in module PrefixIndexing), 33
PrefixIndexing (module), 31
`print_model()` (Model.FactorOracle method), 11
`print_model()` (Model.Model method), 13
`process()` (Query.Query method), 23
`process_priority_query()` (Generator.GenerationHandler method), 25
`process_query()` (Generator.GenerationHandler method), 25
`process_query()` (Generator.Generator method), 29

Q

Query (class in Query), 23
Query (module), 23

R

`receive_query()` (Generator.GenerationHandler method), 26
`receive_query()` (Generator.Generator method), 30
`record_execution_trace()` (Navigator.Navigator method), 16
`reinit_navigation_param()` (ModelNavigator.FactorOracleNavigator method), 21
`reinit_navigation_param()` (Navigator.Navigator method), 17
`relative_to_absolute()` (Query.Query method), 23

S

`similar_backward_context()` (Model.FactorOracle method), 11
`similar_contexts()` (Model.FactorOracle method), 12
`simply_guided_generation()` (Navigator.Navigator method), 17
`simply_guided_navigation()` (ModelNavigator.FactorOracleNavigator method), 21
`simply_guided_navigation()` (Navigator.Navigator method), 17
`start()` (Generator.GenerationHandler method), 27

T

Transforms (module), 35

U

`update_history_and_taboos()` (Navigator.Navigator method), 18

update_performance_time() (Generator.
GenerationHandler method), [27](#)