Naryn - User Manual

September 3, 2019

Naryn helps to efficiently analyze medical records data.

Contents

1 Introduction

Naryn allows efficient access and analysis of medical records that are maintained in a custom database.

Naryn can work under R (as a package) or Python (as a module). The vast majority of the functions and the concepts are shared between the two implementations, yet certain differences still exist and are summarized in a table below. Code examples and function names in this document are presented for R but they can equally run in Python with the interface changes as to the table.

1.1 R vs. Python Interface Differences

	R	Python
Naming Conventions (except for virtual track 'func', which stays unchanged)	`emr_xxx.yyy.zzz`	`xxx_yyy_zzz`

Variables	Defined in global environment:
	`EMR_GROOT`
	`EMR_UROOT`
		Defined in module's environment:
		`_GROOT`
		`_UROOT`

Run-time Variables (available only during track expression evaluation)	`EMR_TIME`	`TIME`

Package / Module Options	Controlled via standard options mechanism:
	`options(emr_xxx.yyy=zzz)`
	`getOption("emr_xxx.yyy")`
		Controlled by module's `CONFIG` variable:
		`CONFIG['xxx_yyy']=zzz`
		`CONFIG['xxx_yyy']`

Data Types (used as function parameters)	`data.frame`
	`list`
	`vector` of strings
	`vector` of numerics
	`NULL`
		`pandas.DataFrame`
		`list`
		`list` of strings
		`numpy.ndarray` of numerics
		`None`

Data Types (return value)	`data.frame`
	`list`
	`vector` of strings
	
	`vector` of numerics, no labels
	`vector` of numerics, with labels
	
	`NULL`
		`pandas.DataFrame`
		`dict`
		`numpy.ndarray` of objects (strings)
		`numpy.ndarray` of numerics
		
		`pandas.DataFrame` with two columns (label, numeric)
		`None`

Database Management	Database is unloaded when the package is detached.	`db_unload()` must be called explicitly to unload the database.
Setting seed for random number generator. **Note:** R and Python use different random generators, results are therefore not reproducible between them.	`set.seed`	`seed`

	R	Python
Track Variables	Variables saved in Python are not visible in R.	Variables saved in R are not visible in Python.
Setting Track Variables	`emr_track.set` creates a directory named `.trackname.var`	`track_set` creates a directory named `.trackname.pyvar`
Named Filters and Virtual Tracks	Named filters and virtuals tracks may be saved along with the rest of R's environment.	`filter_export`, `filter_import`, `vtrack_export`, `vtrack_import` must be explicitly called to save / restore named filters or virtual tracks.
Pattern Matching	`emr_track.ls`, `emr_track.global.ls`, `emr_track.user.ls`, `emr_track.var.ls`, `emr_filter.ls` accept pattern matching parameters. Return: `vector` of strings that match the pattern.	`track_ls`, `track_global_ls`, `track_user_ls`, `track_var_ls`, `filter_ls` do not support pattern matching. Return: `numpy.ndarray` of objects (strings) that contains all the objects (tracks, ...)
Time shift parameter (used in various functions)	`time.shift` is a numeric or a vector of two numerics.	`time_shift` is a numeric or a list of two numerics.
Calculating Distribution	`emr_dist` returns N-dimensional `vector` with labels (dimension names)	`dist` return N-dimensional `numpy.ndarray` without labels.
Calculating Correlation Statistics	`emr_cor`: For N-dimensional binning the returned value `r` may be addressed as: `r$cor[bin1,...,binN,i,j]`, where `i` and `j` are indices of `cor.exprs`.	`cor`: For N-dimensional binning the returned value `r` may be addressed as: `r['cor'][bin1,...,binN,i,j]`, where `i` and `j` are indices of `cor_exprs`.
Others	`emr_annotate`	Not implemented, use `pandas.DataFrame.merge` or `pandas.merge_sorted` instead.
	`emr_traceback`	Not implemented, not applicable.

2 Database

2.1 Global and User Space

Naryn allows accessing the data that resides in *tracks* where each track holds certain type of medical data such as patients' diagnoses or their hemoglobin level at certain points of time. The track files are aggregated in a directory. Before the tracks can be accessed, Naryn needs to establish connection to this directory, also referred as a *global root* or *global space root*. Call `emr_db.init` function to establish the access to the tracks in the global root directory. Optionally `emr_db.init` accepts an additional *user root* (or *user space root*)

directory which can also constains additional tracks.

Even though both global and user root directories may contain track files their designation is different. The global root directory is intended to stay mainly read-only with a notable exception of periodic updates to the existing data which can be performed via `emr_track.addto` function. No tracks are allowed to be deleted from the global root directory. Unlike that user root directory is intended to store volatile data like the results of intermediate calculations. New tracks can be created in both user or global space using `emr_track.import` or `emr_track.create` yet the creation of tracks in the global space is strongly discouraged.

2.2 Load-on-demand vs. Pre-load Modes

`emr_db.init` supports two modes of work - 'load on demand' and 'pre-load'. In 'load on demand' mode tracks are loaded into memory only when they are accessed. Tracks stay in the memory up until R sessions ends or the package is unloaded (Python: since modules cannot be forced to unload, `db_unload` is introduced).

In 'pre-load' mode, all the tracks are pre-loaded into memory making subsequent track access significantly faster. As loaded tracks reside in shared memory, other R sessions running on the same machine may also enjoy significant run-time boost. On the flip side, pre-loading all the tracks prolongs the execution of `emr_db.init` and requires enough memory to accommodate all the data.

Choosing between the two modes depends on the specific needs. While 'load.on.demand=TRUE' seems to be a solid default choice, in an environment where there are frequent short-living R sessions, each accessing a track, one might opt for running a "daemon" - an additional permanent R session. The daemon would pre-load all the tracks in advance and stay alive thus boosting the run-time of the later emerging sessions.

2.3 Maintaining Database

Naryn caches certain data on the disk to maintain fast run-times. In particular two files (`.naryn` and `.ids`) are created in a global root directory and one file (`.ids`) in a user root directory.

`.naryn` file contains a list of all tracks in the current root directory and their last modification dates. This file spares a full root directory rescan when `emr_db.init` is called. The recorded modification dates allow to efficiently synchronize the track changes induced by synchronically running R sessions.

`.ids` file contains available ids that are used to run certain types of *track expression iterators* (see below). The source of these ids comes from a `patients.dob` (i.e. Date Of Birth) track, which must be present in the global root directory before these iterators may be utilized.

Various functions such as `emr_track.import` modify these files according to the changes that DB undergoes (addition / removal / modification of tracks). Thus manual (outside of Naryn) modification, replacement, addition or deletion of track files cause the cache files to go out of sync. Various problems might arise as a consequence, such as run-time errors, out-dated data from modified tracks and sub-optimal run-time performance.

Manual modifications of the database files can still be performed, yet they must be ratified by running `emr_db.reload`.

2.4 Tracks

Each track is stored in a binary file with *.nrtrack* file extention. One of the two internal formats, *dense* or *sparse*, is automatically selected during the track creation. The choice of the exact format is based on the optimal run-time performance.

2.4.1 Records and References

Track is a data structure that stores a set of records of `(id, time, ref, numeric value)` type. For example, hemoglobin level of patients can be stored in this way, where `id` would be the id of the patient and *time* would indicate the moment when the blood test was made. Another track can contain the code of the laboratory which carried out the test. If the times of the records from the two tracks match, one would conclude which lab performed the given test.

Time resolution is always in hours. It might happen that two different blood tests are carried out by two different labs for the same patient at the same hour. Assuming that each lab has certain bias due to different

equipment used, the reads of the hemoglobin might come out different. Since both of the tests are carried out at exactly the same hour it will be impossible later to link each result to the lab that performed it.

In those cases when two or more values share identical `id` and `time` Naryn requires them to use then different `ref` (*references*). Reference is an integer number in the range of [-1, 254], which when no time collision occurs is normally set to -1. However in cases of ambiguity it can give additional resolution to the time. In our blood example the results of the first lab could have been recorded with `ref = 0` wherever the second lab would do it with `ref = 1`. This way the two hemoglobin readings could later be separated and correctly linked to their originating labs.

2.4.2 Categorical and Quantitative Tracks

Tracks store numerical values assigned to the patients and times. The numerical data however can have different meaning and hence impose different set of operations to be applied to it. Laboratory codes, diagnosis codes, binary information such as date of birth or doctor visits are one type of data which we call *categorical*. Another type of data indicate usually the readings of different instruments such as the heartbeat rate or glucose level. This type of data is called *quantitative*.

The operations that can be applied to both of these types can be very different. One might want to search for the specific diagnosis code, yet it makes little sense to search for the very specific heartbeat rate, say "68". On contrary hearbeat rate readings from different times can be averaged or a mean value might be calculated - something that has no meaning in case of categorical data.

During the track creation one must specify the type of the track: categorical or quantitative. Various operations that can be later applied to the track are bound to the track type.

2.4.3 Track Variables

Track statistics, results of time-consuming per-track calculations, historical data and any other data in arbitrary format can be stored in a track's supplementary data in the form of track variables. Track variable can be retrieved, added, modified or deleted using `emr_track.var.get`, `emr_track.var.set`, `emr_track.var.rm` functions. List of track variables can be retrieved using `emr_track.var.ls` function.

Note: track variables created in R are not visible in Python and vice versa.

2.4.4 Subsets

The analysis of data often involves dividing the data to train and test sets. Naryn allows to subset the data via `emr_db.subset` function. `emr_db.subset` accepts a list of ids or samples the ids randomally. These ids constitute the subset. The ids that are not in the subset are skipped by all the *iterators*, *filters* and various functions.

One may think of a subset as an additonal layer, a "viewport", that filters out some of the ids. Some lower-level functions such as `emr_track.info` or `emr_track.unique` ignore the subsets. Same applies to `percentile.*` functions of the virtual tracks.

3 Accessing the Data

3.1 Track Expressions

3.1.1 Introduction

Track expression allows to retrieve numerical data that is recorded in the tracks. Track expressions are widely used in various functions (`emr_screen`, `emr_extract`, `emr_dist`, ...).

Track expression is a character string that closely resembles a valid R/Python expression. Just like any other R/Python expression it may include conditions, function calls and variables defined beforehand. `"1 > 2"`, `"mean(1:10)"` and `"myvar < 17"` are all valid track expressions. Unlike regular R/Python expressions track expression might also contain track names and / or *virtual track* names.

To understand how the track expression allows the access to the tracks we must explain how the track expression gets evaluated.

Every track expression is accompanied by an *iterator* that produces a set of *id-time points* of `(id, time, ref)` type. For each each iterator point the track expression is evaluated. The value of the track expression `"mean(1:10)"` is constant regardless the iterator point. However the track expression might contain a track name `mytrack`, like: `"mytrack * 3"`. Naryn recognizes then that `mytrack` is not a regular R/Python variable but rather a track name. A new *run-time track variable* named `mytrack` is added then to R environment (or Python module local dictionary). For each iterator point this variable is assigned the value of the track that matches `(id, time, ref)` (or NaN if no matching value exists in the track). Once `mytrack` is assigned the corresponding value, the track expression is evaluated in R/Python.

3.1.2 Run-time Track Variable is a Vector

To boost the performance of the track expression evaluation, run-time track variables are actually defined as vectors in R rather than scalars. The result of the evaluation is expected to be also a vector of a similar size. One should always keep in his mind the vectorial notation and write the track expressions accordingly.

For example, at first glance a track expression `"min(mytrack, 10)"` seems to be perfectly fine. However the evaluation of this expression produces always a scalar, i.e. a signle number even if `mytrack` is actually a vector. The way to correct the specific track expression so that it works on vectors, is to use `pmin` function instead of `min`.

Python
Similarly to R, a track variable in Python is not a scalar but rather an instance of `numpy.ndarray`. The evaluation of a track expression must therefore produce a `numpy.ndarray` as well. Various operations on numpy arrays indeed work the same way as with scalars, however logical operations require different syntax. For instance,
`screen("mytrack1 > 1 and mytrack2 < 2", iterator="mytrack1")`
will produce an error given that `mytrack1` and `mytrack2` are numpy arrays. The correct way to write the expression is:
`screen("(mytrack1 > 1) & (mytrack2 < 2)", iterator="mytrack1")`

One may coerce the track variable to behave like a scalar: by setting `emr_eval.buf.size` option to `1` (see Appendix for more details). Beware though that this might take its heavy toll on run-time.

3.1.3 Matching Reference in the Track Expression

If the track expression contains a track (or virtual track) name, then the values from the track are fetched one-by-one into the identically named R variable based on `id`, `time` and `ref` of the iterator point. If however `ref` of the iterator point equals to `-1`, we treat it as a "wildcard": matching is required then only for `id` and `time`.

"Wildcard" reference in the iterator might create a new issue: more than one track value might match then a single iterator point. In this case the value placed in the track variable (e.g. `mytrack`) depends on the type of the track. If the track is categorical the track variable is set to `-1`, otherwise it is set to the average of all matching values.

3.1.4 Virtual Tracks

So far we have shown that in some situations `mytrack` variable can be set to the average of the matching track values. But what if we do not want to average the values but rather pick up the maximal, minimal or median value? What if we want to use the percentile of a track value rather than the value itself? And maybe we even want to alter the time of the iterator point: shift it or expand to a time window and by that look at the different set of track values? For instance: given an iterator point we might want to know what was the maximal level of glucose during the last year that preceeded the time of the point.

This is where virtual tracks come in use.

Virtual track is a named set of rules that describe how the track should be proceeded, and how the time of the iterator point should be modified. Virtual tracks are created by `emr_vtrack.create` function:

```
emr_vtrack.create("annual_glucose", src="glucose_track", func="quantile",
                  param=0.5, time.shift=c(-365*24, 0))
```

This call creates a new virtual track named `annual_glucose` based on the underlying physical *source track* `glucose_track`. For each iterator point with time `T` we look at values of `glucose_track` in the time window of `[T-365*24,T]`, i.e. one year prior to `T`. We calculate then the median over the values (`func="quantile"`, `param=0.5`).

There is a rich set of various functions besides "quantile" that can be applied to the track values. Some of these functions can be used only with categorical tracks, other ones - only with quantitative tracks and some functions can be applied to both types of the track. Please refer the documentation of `emr_vtrack.create`.

Once a virtual track is created it can be used in a track expression:

```
emr_extract("annual_glucose", iterator=list(365*24, 'date_of_birth_track'))
```

This would give us a median of an annual glucose level in year-steps starting from the patient's birthday. (This example makes use of an *Extended Beat Iterator* that would be expalained later.)

Let's expand our example further and ignore in our calculations the glucose readings that had been made within a week after steroids had been prescribed. We can use an additional `filter` parameter to do that.

```
emr_filter.create("steroids_filter", "steroids_track", time.shift=c(-24*7, 0))
emr_vtrack.create("annual_glucose", src="glucose_track", func="quantile",
                  param=0.5, time.shift=c(-365*24,0), filter="!steroids_filter")
emr_extract("annual_glucose", iterator=list(365*24, "date_of_birth_track"))
```

Filter is applied to the ID-Time points of the source track (e.g. `glucose_track` in our example). The virtual track function (`quantile`, ...) is applied then only to the points that pass the filter. The concept of filters is explained extensively in a separate chapter.

Virtual tracks allow also to remap the patient ids. This is done via `id.map` parameter which accepts a data frame that defines the id mapping. Remapping ids might be useful if family ties are explored. For example, instead of glucose level of the patient we are interested to check the glucose level of one of his family members.

3.2 Iterators

So far we have discussed the track expressions and how they are evaluated given the iterator point. In this section we will show how the iterator points are generated.

An iterator is defined via `iterator` parameter. There are a few types of iterators such as *track iterator*, *beat iterator*, etc. The type determines which points are generated by the iterator. The information about each type is listed below.

Iterator is always accompanied by four additional parameters: `stime`, `etime`, `keepref` and `filter`. `stime` and `etime` bind the time scope of the iterator: the points that the iterator generates lie always within these boundaries. The effect of `keepref=T` depends on the iterator type. However for all the iterator types if `keepref=F` the reference of all the iterator points is set to `-1`. `filter` parameter sets the iterator filter which is discussed thoroughly later in the document in a separate chapter.

3.2.1 Track Iterator

Track iterator returns the points (including the reference) from the specified track. Track name is specified as a string.

If `keepref=F` the reference of each point is set to `-1`.

Example:

```
# Returns the level of glucose one hour after the insulin shot was made
emr_vtrack.create("glucose", "glucose_track", func="avg", time.shift=1)
emr_extract("glucose", iterator="insulin_shot_track")
```

3.2.2 Id-Time Points Iterator

Id-Time points iterator generates points from an *id-time points table* (see: Appendix). If `keepref=F` the reference of each point is set to `-1`.

Example:

```
# Returns the level of glucose one hour after the insulin shot was made
emr_vtrack.create("glucose", "glucose_track", func="avg", time.shift=1)
r <- emr_extract("insulin_shot_track")  # <-- implicit iterator is used here
emr_extract("glucose", iterator=r)
```

3.2.3 Ids Iterator

Ids iterator generates points with ids taken from an *ids table* (see: Appendix) and times that run from `stime` to `etime` with a step of 1.

If `keepref=T` for each id-time pair the iterator generates 255 points with references running from `0` to `254`. If `keepref=F` only one point is generated for the given id and time, and its reference is set to `-1`.

Example:

```
# Returns the level of glucose for each hour in year 2016 for ids 2 and 5
stime <- emr_date2time(1, 1, 2016, 0)
etime <- emr_date2time(31, 12, 2016, 23)
emr_extract("glucose", iterator=data.frame(id=c(2,5)), stime=stime, etime=etime)
```

3.2.4 Time Intervals Iterator

Time intervals iterator generates points for all the ids that appear in 'patients.dob' track with times taken from a *time intervals table* (see: Appendix). Each time starts at the beginning of the time interval and runs to the end of it with a step of 1. That being said the points that lie outside of `[stime, etime]` range are skipped.

If `keepref=T` for each id-time pair the iterator generates 255 points with references running from `0` to `254`. If `keepref=F` only one point is generated for the given id and time, and its reference is set to `-1`.

Example:

```
# Returns the level of hangover for all patients the next day after New Year Eve
# for the years 2015 and 2016
stime1 <- emr_date2time(1, 1, 2015, 0)
etime1 <- emr_date2time(1, 1, 2015, 23)
stime2 <- emr_date2time(1, 1, 2016, 0)
etime2 <- emr_date2time(1, 1, 2016, 23)
emr_extract("alcohol_level_track", iterator=data.frame(stime=c(stime1, stime2),
            etime=c(etime1, etime2)))
```

3.2.5 Id-Time Intervals Iterator

Id-Time intervals iterator generates for each id points that cover `['stime', 'etime']` time range as specified in *id-time intervals table* (see: Appendix). Each time starts at the beginning of the time interval and runs to the end of it with a step of 1. That being said the points that lie outside of `[stime, etime]` range are skipped.

If `keepref=T` for each id-time pair the iterator generates 255 points with references running from `0` to `254`. If `keepref=F` only one point is generated for the given id and time, and its reference is set to `-1`.

3.2.6 Beat Iterator

Beat Iterator generates a "time beat" at the given period for each id that appear in 'patients.dob' track. The period is given always in hours.

Example:

```
emr_extract("glucose_track", iterator=10, stime=1000, etime=2000)
```

This will create a beat iterator with a period of 10 hours starting at `stime` up until `etime` is reached. If, for example, `stime` equals `1000` then the beat iterator will create for each id iterator points at times: 1000, 1010, 1020, ...

If `keepref=T` for each id-time pair the iterator generates 255 points with references running from `0` to `254`. If `keepref=F` only one point is generated for the given id and time, and its reference is set to `-1`.

3.2.7 Extended Beat Iterator

Extended beat iterator is as its name suggests a variation on the beat iterator. It works by the same principle of creating time points with the given period however instead of basing the times count on `stime` it accepts an additional parameter - a track or a *Id-Time Points table* - that instructs what should be the initial time point for each of the ids. The two parameters (period and mapping) should come in a list. Each id is required to appear only once and if a certain id does not appear at all, it is skipped by the iterator.

Anyhow points that lie outside of `[stime, etime]` range are not generated.

Example:

```
# Returns the maximal weight of patients at one year span starting from their birthdays
emr_vtrack.create("weight", "weight_track", func="max", time.shift=c(0, 24*365))
emr_extract("weight", iterator=list(24*365, "birthday_track"), stime=1000, etime=2000)
```

3.2.8 Implicit Iterator

The iterator is set implicitly if its value remains `NULL` (which is the default). In that case the track expression is analyzed and searched for track names. If all the track variables or virtual track variables point to the same track, this track is used as a source for a track iterator. If more then one track appears in the track expression, an error message is printed out notifying ambiguity.

3.2.9 Revealing Current Iterator Time

During the evaluation of a track expression one can access a specially defined variable named `EMR_TIME` (Python: `TIME`). This variable contains a vector (`numpy.ndarray` in Python) of current iterator times. The length of the vector matches the length of the track variable (which is a vector too).

Note that some values in `EMR_TIME` might be set 0. Skip those intervals and the values of the track variables at the corresponding indices.

```
# Returns times of the current iterator as a day of month
emr_extract("emr_time2dayofmonth(EMR_TIME)", iterator = "sparse_track")
```

3.3 Filters

Filter is used to approve / reject an ID-Time point. It can be applied to an iterator, in which case the iterator points are required to be approved by the filter before they are passed further to the track expression. Filter may also be used by a virtual track. In this case the virtual track function (see `func` parameter of `emr_vitrack.create`) is applied only to the points from the source track (`src` parameter) that pass the filter.

Filter has a form of a logical expression consisting of *named* or *unnamed elementary filters* (the "building bricks" of the filter) connected with the logical operators: `&`, `|`, `!` (`and`, `or` and `not` in Python) and brackets `()`.

3.3.1 Named Filters

Suppose we are interested in hemoglobin levels of patients who were prescribed either drugX or drugY but not drugZ within a time window of one week before the test. Assume that drugX, drugY and drugZ are residing each in its separate track. Without filters we would need to call `emr_extract` four times, store

potentially huge data frame results in the memory and finally merge the tables within R while caring about time windows. With filters we can do it much easier:

```
emr_filter.create("filterX", "drugX", time.shift = c(24 * 7, 0))
emr_filter.create("filterY", "drugY", time.shift = c(24 * 7, 0))
emr_filter.create("filterZ", "drugZ", time.shift = c(24 * 7, 0))
emr_extract("hemoglobin", filter = "(filterX | filterY) & !filterZ")
```

Python
Filter with logical conditions will use Python's notation like:
`extract("hemoglobin", filter = "(filterX or filterY) and not filterZ")`

Each call to `emr_filter.create` creates a *named elementary filter* (or simply: named filter) with a unique name. The named filter can then be used in `filter` parameter of an iterator and be combined with other named filters using the logical operators.

3.3.2 Other Objects within Filters

In our previous example we created three named filters based on three tracks. If time window was not required, we could have used the names of the tracks directly in the filter, like: `filter = "(drugX | drugY) & !drugZ"`.

In addition to track names other types of objects can be used within the filter. These are: *Id-Time Points Table*, *Ids Table*, *Time Intervals Table* and *Id-Time Intervals Table* (see *Appendix* for the format of these tables). When used in the filter the object should be constructed in advanced and be referred by its name. "In place" construction (aka: `filter = "data.frame(...)"` is not allowed.

3.3.3 Managing Reference in Filters

The ID-Time Point embeds within itself a reference value. Named filters allow to specify whether the reference should be used for matching or not. When `keepref=TRUE` is set within `emr_filter.create`, the candidate point's reference is matched with the filter's reference. Otherwise the references are ignored.

It is important to remember that references are always ignored when any object but a named filter is used within a filter. For instance, if `filter = "drug"` and `drug` is a name of a track (and not a name of a named filter), then the references will be ignored during the matching. To ensure the filter matches the references of `drug` track, one must define a named filter with `keepref=TRUE` parameter:

```
emr_filter.create("drug_filter", "drug", keepref=TRUE)
emr_extract(my.track.expression, filter="drug_filter", keepref=TRUE)
```

4 Advanced Naryn

4.1 Random Algorithms

Various functions in the library such as `emr_quantiles` make use of pseudo-random number generator. Each time the function is invoked a unique series of random numbers is issued. Hence two identical calls might produce different results. To guarantee reproducible results call `set.seed` (Python: textttseed) before invoking the function.

Note: R and Python implementations of Naryn use different pseudo-random number generator algorithms. Sadly it means that the result achieved in R cannot be reproducible in Python if random is used, even if identical seed is shared between the two platforms.

4.2 Multitasking

To boost the run time performance various functions in the library support multitasking mode, i.e. parallel computation by several concurrent processes. Multitasking is not invoked immediately: approximately 0.3

seconds from the function launch the actual progress is measured and total run-time is estimated. If the estimated run-time exceeds the limit (currently: 2 seconds), multitasking kicks in.

The number of processes launched in the multitasking mode depends on the total run-time estimation (longer run-time will use more processes) and the values of `emr_min.processes` and `emr_max.processes` R options. In any case the number of processes never exceeds the number of CPU cores available.

Multitasking can significantly boost the performance however it utilizes more CPU. When CPU utilization is the priority it is advisable to switch off multitasking by setting `emr_multitasking` R option to `FALSE`.

In addition to increased CPU usage multitasking might also alter the behavior of functions that return ID-Time points such as `emr_extract` and `emr_screen`. When multitasking is not invoked these functions return the results always sorted by ID, time and reference. In multitasking mode however the result might come out unsorted. Moreover subsequent calls might return results reshuffled differently. One might use `sort` parameter in these functions to ensure the points come out sorted. Please bear in mind that sorting the results takes its toll especially on particularly large data frames. That's why by default `sort` is set to `FALSE`.

5 Appendix

5.1 Options

Naryn supports the following options. The options can be set/examined via R's `options` and `getOption`.

(Use `CONFIG['option_name']` to control the module options in Python. Please mind as well Python's name convention: R's `emr_xxx.yyy` option will change its name to `xxx_yyy`.)

Option	Default Value	Description
`emr_multitasking`	`TRUE`	Should the multitasking be allowed?
`emr_min.processes`	`8`	Minimal number of processes launched when multitasking is invoked.
`emr_max.processes`	`20`	Maximal number of processes launched when multitasking is invoked.
`emr_max.data.size`	`10000000`	Maximal size of data sets (rows of a data frame, length of a vector, ...) stored in memory. Prevents excessive memory usage.
`emr_eval.buf.size`	`1000`	Size of the track expression evaluation buffer.
`emr_warning.itr.no.filter.size`	`100000`	Threshold above which "beat iterator used without filter" warning is issued.

5.2 Common Table Formats

5.3 Id-Time Points Table

Id-Time Points table is a data frame having two first columns named 'id' and 'time'. References might be specified by a third column named 'ref'. If 'ref' column is missing or named differently references are set to `-1`. Additional columns, if presented, are ignored.

5.4 Id-Time Values Table

Id-Time Values table is an extension of *Id-Time Points table* with an additional column named 'value'. Additional columns, if presented, are ignored.

5.5 Ids Table

Ids table is a data frame having the first column named 'id'. Each id must appear only once. Additional columns of the data frame, if presented, are ignored.

5.6 Time Intervals Table

Time Intervals table is a data frame having two first columns named 'stime' and 'etime' (i.e. start time and end time). Additional columns, if presented, are ignored.

5.7 Id-Time Intervals Table

Id-Time Intervals table is a data frame having three first columns named 'id', 'stime' and 'etime' (i.e. start time and end time). Additional columns, if presented, are ignored.