MAD-NG

Release 0.9.6

Laurent Deniau

Aug 30, 2022

CONTENTS:

CHAPTER ONE

SEQUENCES

The MAD Sequences are objects convenient to describe accelerators lattices built from a *list* of elements with increasing `s`-positions. The sequences are also containers that provide fast access to their elements by referring to their indexes, `s`-positions, or (mangled) names, or by running iterators constrained with ranges and predicates. The `sequence` object is the *root object* of sequences that store information relative to lattices.

The `sequence` module extends the *typeid* module with the `is_sequence` function, which returns `true` if its argument is a `sequence` object, `false` otherwise.

1.1 Attributes

The `sequence` object provides the following attributes:

l
: A *number* specifying the length of the sequence `[m]`. A `nil` will be replaced by the computed lattice length. A value greater or equal to the computed lattice length will be used to place the `$end` marker. Other values will raise an error. (default: `nil`).

dir
: A *number* holding one of `1` (forward) or `-1` (backward) and specifying the direction of the sequence.[1] (default:~ `1`)

refer
: A *string* holding one of `"entry"`, `"centre"` or return true `"exit"` to specify the default reference position in the elements to use for their placement. An element can override it with its `refpos` attribute, see *element positions* details. (default: `nil` $\equiv$ `"centre"`).

owner
: A *logical* specifying if an *empty* sequence is a view with no data `(owner ~= true)`, or a sequence holding data `(owner == true)`. (default: `nil`)

minlen
: A *number* specifying the minimal length `[m]` to generate *implicit* drifts between elements in s-iterators generated by the method `:siter`. This attribute is automatically set to 10^{-6} m when a sequence is created within the MADX environment. (default: `nil`)

beam
: An attached `beam`. (default: `nil`)

Warning: the following private and read-only attributes are present in all sequences and should *never be used, set or changed*; breaking this rule would lead to an *undefined behavior*:

[1] This is equivalent to the MAD-X bv flag.

__dat
A *table* containing all the private data of sequences.

__cycle
A *reference* to the element registered with the `:cycle` method. (default: `nil`)

1.2 Methods

The `sequence` object provides the following methods:

elem
A *method* `(idx)` returning the element stored at the positive index `idx` in the sequence, or `nil`.

spos
A *method* `(idx)` returning the s-position at the entry of the element stored at the positive index `idx` in the sequence, or `nil`.

upos
A *method* `(idx)` returning the s-position at the user-defined `refpos` offset of the element stored at the positive index `idx` in the sequence, or `nil`.

ds
A *method* `(idx)` returning the length of the element stored at the positive index `idx` in the sequence, or `nil`.

align
A *method* `(idx)` returning a *set* specifying the misalignment of the element stored at the positive index `idx` in the sequence, or `nil`.

index
A *method* `(idx)` returning a positive index, or `nil`. If `idx` is negative, it is reflected versus the size of the sequence, e.g. `-1` becomes `#self`, the index of the `$end` marker.

name_of
A *method* `(idx, [ref])` returning a *string* corresponding to the (mangled) name of the element at the index `idx` or `nil`. An element name appearing more than once in the sequence will be mangled with an absolute count, e.g. `mq[3]`, or a relative count versus the optional reference element `ref` determined by `:index_of`, e.g. `mq{-2}`.

index_of
A *method* `(a, [ref], [dir])` returning a *number* corresponding to the positive index of the element determined by the first argument or `nil`. If `a` is a *number* (or a *string* representing a *number*), it is interpreted as the s-position of an element and returned as a second *number*. If `a` is a *string*, it is interpreted as the (mangled) name of an element as returned by `:name_of`. Finally, `a` can be a *reference* to an element to search for. The argument `ref` (default: `nil`) specifies the reference element determined by `:index_of(ref)` to use for relative s-positions, for decoding mangled names with relative counts, or as the element to start searching from. The argument `dir` (default: `1`) specifies the direction of the search with values `1` (forward), `-1` (backward), or `0` (no direction). The `dir=0` case may return an index at half-integer if `a` is interpreted as an s-position pointing to an *implicit drift*.

range_of
A *method* `([rng], [ref], [dir])` returning three *numbers* corresponding to the positive indexes *start* and *end* of the range and its direction *dir*, or `nil` for an empty range. If `rng` is omitted, it returns `1`, `#self`, `1`, or `#self`, `1`, `-1` if `dir` is negative. If `rng` is a *number* or a *string* with no `'/'` separator, it is interpreted as both *start* and *end* and determined by `index_of`. If `rng` is a *string* containing the separator `'/'`, it is split in two *strings* interpreted as *start* and *end*, both determined by `:index_of`. If `rng` is a *list*, it will be interpreted as {*start*, *end*, `[ref]`, `[dir]`}, both determined by `:index_of`, unless `ref` equals `'idx'` then both are determined by `:index` (i.e. a *number* is interpreted as an index instead of a s-position). The arguments `ref` (default: `nil`)

and `dir` (default: `1`) are forwarded to all invocations of `:index_of` with a higher precedence than ones in the *list* `rng`, and a runtime error is raised if the method returns `nil`, i.e. to disambiguate between a valid empty range and an invalid range.

length_of
A *method* `([rng], [ntrn], [dir])` returning a *number* specifying the length of the range optionally including `ntrn` extra turns (default: `0`), and calculated from the indexes returned by `:range_of([rng], nil, [dir])`.

iter
A *method* `([rng], [ntrn], [dir])` returning an iterator over the sequence elements. The optional range is determined by TT{:range_of(rng, [dir])}, optionally including `ntrn` turns (default: `0`). The optional direction `dir` specifies the forward `1` or the backward `-1` direction of the iterator. If `rng` is not provided and the ?sequence? is cycled, the *start* and *end* indexes are determined by `:index_of(self.__cycle)`. When used with a generic `for` loop, the iterator returns at each element: its index, the element itself, its *s*-position over the running loop and its signed length depending on the direction.

siter
A *method* `([rng], [ntrn], [dir])` returning an *s*-iterator over the sequence elements. The optional range is determined by `:range_of([rng], nil, [dir])`, optionally including `ntrn` turns (default: `0`). The optional direction `dir` specifies the forward `1` or the backward `-1` direction of the iterator. When used with a generic `for` loop, the iterator returns at each iteration: its index, the element itself or an *implicit* `drift`, its *s*-position over the running loop and its signed length depending on the direction. Each *implicit* drift is built on-the-fly by the iterator with a length equal to the gap between the elements surrounding it and a half-integer index equal to the average of their indexes. The length of *implicit* drifts is bounded by the maximum between the sequence attribute `minlen` and the `minlen` from the *constant* module.

foreach
A *method* `(act, [rng], [sel], [not])` returning the sequence itself after applying the action `act` on the selected elements. If `act` is a *set* representing the arguments in the packed form, the missing arguments will be extracted from the attributes `action`, `range`, `select` and `default`. The action `act` must be a *callable* `(elm, idx, [midx])` applied to an element passed as first argument and its index as second argument, the optional third argument being the index of the main element in case `elm` is a sub-element. The optional range is used to generate the loop iterator `:iter([rng])`. The optional selector `sel` is a *callable* `(elm, idx, [midx])` predicate selecting eligible elements for the action using the same arguments. The selector `sel` can be specified in other ways, see *element selections* for details. The optional *logical* `not` (default: `false`) indicates how to interpret default selection, as *all* or *none*, depending on the semantic of the action.[2]

select
A *method* `([flg], [rng], [sel], [not])` returning the sequence itself after applying the action `:select([flg])` to the elements using `:foreach(act, [rng], [sel], [not])`. By default sequence have all their elements deselected with only the `$end` marker `observed`.

deselect
A *method* `([flg], [rng], [sel], [not])` returning the sequence itself after applying the action `:deselect([flg])` to the elements using `:foreach(act, [rng], [sel], [not])`. By default sequence have all their elements deselected with only the `$end` marker `observed`.

filter
A *method* `([rng], [sel], [not])` returning a *list* containing the positive indexes of the elements determined by `:foreach(act, [rng], [sel], [not])`, and its size. The *logical* `sel.subelem` specifies to select sub-elements too, and the *list* may contain non-integer indexes encoding their main element index added to their relative position, i.e. `midx.sat`. The builtin *function* `math.modf(num)` allows to retrieve easily the main element `midx` and the sub-element `sat`, e.g. `midx,sat = math.modf(val)`.

install
A *method* `(elm, [rng], [sel], [cmp])` returning the sequence itself after installing the elements in the

[2] For example, the `:remove` method needs `not=true` to *not* remove all elements if no selector is provided.

list `elm` at their *element positions*; unless `from="selected"` is defined meaning multiple installations at positions relative to each element determined by the method `:filter([rng], [sel], true)`. The *logical* `sel.subelem` is ignored. If the arguments are passed in the packed form, the extra attribute `elements` will be used as a replacement for the argument `elm`. The *logical* `elm.subelem` specifies to install elements with s-position falling inside sequence elements as sub-elements, and set their `sat` attribute accordingly. The optional *callable* `cmp(elmspos, spos[idx])` (default: `"<"`) is used to search for the s-position of the installation, where equal s-position are installed after (i.e. before with `"<="`), see `bsearch` from the *utility* module for details. The *implicit* drifts are checked after each element installation.

replace
A *method* `(elm, [rng], [sel])` returning the *list* of replaced elements by the elements in the *list* `elm` placed at their *element positions*, and the *list* of their respective indexes, both determined by `:filter([rng], [sel], true)`. The *list* `elm` cannot contain instances of `sequence` or `bline` elements and will be recycled as many times as needed to replace all selected elements. If the arguments are passed in the packed form, the extra attribute `elements` will be used as a replacement for the argument `elm`. The *logical* `sel.subelem` specifies to replace selected sub-elements too and set their `sat` attribute to the same value. The *implicit* drifts are checked only once all elements have been replaced.

remove
A *method* `([rng], [sel])` returning the *list* of removed elements and the *list* of their respective indexes, both determined by `:filter([rng], [sel], true)`. The *logical* `sel.subelem` specifies to remove selected sub-elements too.

move
A *method* `([rng], [sel])` returning the sequence itself after updating the *element positions* at the indexes determined by `:filter([rng], [sel], true)`. The *logical* `sel.subelem` is ignored. The elements must keep their order in the sequence and surrounding *implicit* drifts are checked only once all elements have been moved.[3]

misalign
A *method* `(algn, [rng], [sel])` returning the sequence itself after setting the *element misalignments* from `algn` at the indexes determined by `:filter([rng], [sel], true)`. If `algn` is a *mappable*, it will be used to misalign the filtered elements. If `algn` is a *iterable*, it will be accessed using the filtered elements indexes to retrieve their specific misalignment. If `algn` is a *callable* `(idx)`, it will be invoked for each filtered element with their index as solely argument to retrieve their specific misalignment.

reflect
A *method* `([name])` returning a new sequence from the sequence reversed, and named from the optional *string* `name` (default: `self.name..'_rev'`).

cycle
A *method* `(a)` returning the sequence itself after checking that `a` is a valid reference using `:index_of(a)`, and storing it in the `__cycle` attribute, itself erased by the methods editing the sequence like `:install`, `:replace`, `:remove`, `:share`, and `:unique`.

share
A *method* `(seq2)` returning the *list* of elements removed from the `seq2` and the *list* of their respective indexes, and replaced by the elements from the sequence with the same name when they are unique in both sequences.

unique
A *method* `([fmt])` returning the sequence itself after replacing all non-unique elements by new instances sharing the same parents. The optional `fmt` must be a *callable* `(name, cnt, idx)` that returns the mangled name of the new instance build from the element `name`, its count `cnt` and its index `idx` in the sequence. If the optional `fmt` is a *string*, the mangling *callable* is built by binding `fmt` as first argument to the function `string.format` from the standard library, see Lua 5.2 §6.4 for details.

[3] Updating directly the positions attributes of an element has no effect.

publish
A *method* `(env, [keep])` returning the sequence after publishing all its elements in the environment `env`. If the *logical* `keep` is `true`, the method will preserve existing elements from being overridden. This method is automatically invoked with `keep=true` when sequences are created within the `MADX` environment.

copy
A *method* `([name], [owner])` returning a new sequence from a copy of `self`, with the optional `name` and the optional attribute `owner` set. If the sequence is a view, so will be the copy unless `owner == true`.

is_view
A *method* () returning `true` if the sequence is a view over another sequence data, `false` otherwise.

set_readonly
Set the sequence as read-only, including its columns.

save_flags
A *method* `([flgs])` saving the flags of all the elements to the optional *iterable* `flgs` (default: `{}`) and return it.

restore_flags
A *method* `(flgs)` restoring the flags of all the elements from the *iterable* `flgs`. The indexes of the flags must match the indexes of the elements in the sequence.

dumpseq
A *method* `([fil], [info])`} displaying on the optional file `fil` (default: `io.stdout`) information related to the position and length of the elements. Useful to identify negative drifts and badly positioned elements. The optional argument `info` indicates to display extra information like elements misalignments.

check_sequ
A *method* () checking the integrity of the sequence and its dictionary, for debugging purpose only.

1.3 Metamethods

The `sequence` object provides the following metamethods:

__len
A *method* () called by the length operator `#` to return the size of the sequence, i.e. the number of elements stored including the `"\$start"` and `"\$end"` markers.

__index
A *method* `(key)` called by the indexing operator `[key]` to return the *value* of an attribute determined by *key*. The *key* is interpreted differently depending on its type with the following precedence: 1. A *number* is interpreted as an element index and returns the element or `nil`. #. Other *key* types are interpreted as *object* attributes subject to object model lookup. #. If the *value* associated with *key* is `nil`, then *key* is interpreted as an element name and returns either the element or an *iterable* on the elements with the same name.[4] #. Otherwise returns `nil`.

__newindex
A *method* `(key, val)` called by the assignment operator `[key]=val` to create new attributes for the pairs (*key*, *value*). If *key* is a *number* specifying the index or a *string* specifying the name of an existing element, the following error is raised: `"invalid sequence write access (use replace method)"`

__init
A *method* () called by the constructor to compute the elements positions.

__copy
A *method* () similar to the `:copy` *method*.

[4] An *iterable* supports the length operator `#`, the indexing operator `[]` and generic `for` loops with `ipairs`.

The following attribute is stored with metamethods in the metatable, but has different purpose:

__sequ A unique private *reference* that characterizes sequences.

1.4 Sequences creation

During its creation as an *object*, a sequence can defined its attributes as any object, and the *list* of its elements that must form a *sequence* of increasing s-positions. When subsequences are part of this *list*, they are replaced by their respective elements as a sequence *element* cannot be present inside other sequences. If the length of the sequence is not provided, it will be computed and set automatically. During their creation, sequences compute the s-positions of their elements as described in the section *element positions*, and check for overlapping elements that would raise a "negative drift" runtime error.

The following example shows how to create a sequence form a *list* of elements and subsequences:

```
local sequence, drift, marker in MAD.element
local df, mk = drift 'df' {l=1}, marker 'mk' {}
local seq = sequence 'seq' {
df 'df1' {}, mk 'mk1' {},
sequence {
   sequence { mk 'mk0' {} },
   df 'df.s' {}, mk 'mk.s' {}
},
df 'df2' {}, mk 'mk2' {},
} :dumpseq()
```

Displays

```
sequence: seq, l=3
idx  kind     name         l          dl       spos       upos     uds
001  marker   (*$start*)   0.000       0       0.000      0.000    0.000
002  drift    df1          1.000       0       0.000      0.500    0.500
003  marker   mk1          0.000       0       1.000      1.000    0.000
004  marker   mk0          0.000       0       1.000      1.000    0.000
005  drift    df.s         1.000       0       1.000      1.500    0.500
006  marker   mk.s         0.000       0       2.000      2.000    0.000
007  drift    df2          1.000       0       2.000      2.500    0.500
008  marker   mk2          0.000       0       3.000      3.000    0.000
009  marker   (*$end*)     0.000       0       3.000      3.000    0.000
```

1.5 Element positions

A sequence looks at the following attributes of an element, including sub-sequences, when installing it, *and only at that time*, to determine its position:

at

A *number* holding the position in [m] of the element in the sequence relative to the position specified by the `from` attribute.

from

A *string* holding one of `"start"`, `"prev"`, `"next"`, `"end"` or `"selected"`, or the (mangled) name of another element to use as the reference position, or a *number* holding a position in [m] from the start of the sequence. (default: `"start"` if `at`≥ 0, `"end"` if `at`< 0, and `"prev"` otherwise)

refpos
A *string* holding one of `"entry"`, `"centre"` or `"exit"`, or the (mangled) name of a sequence sub-element to use as the reference position, or a *number* specifying a position [m] from the start of the element, all of them resulting in an offset to substract to the `at` attribute to find the s-position of the element entry. (default: `nil` $\equiv$ `self.refer`).

shared
A *logical* specifying if an element is used at different positions in the same sequence definition, i.e. shared multiple times, through temporary instances to store the many `at` and `from` attributes needed to specify its positions. Once built, the sequence will drop these temporary instances in favor of their common parent, i.e. the original shared element.

Warning:
The `at` and `from` attributes are not considered as intrinsic properties of the elements and are used only once during installation. Any reuse of these attributes is the responsibility of the user, including the consistency between `at` and `from` after updates.

1.6 Element selections

The element selection in sequence use predicates in combination with iterators. The sequence iterator manages the range of elements where to apply the selection, while the predicate says if an element in this range is illegible for the selection. In order to ease the use of methods based on the `:foreach` method, the selector predicate `sel` can be built from different types of information provided in a *set* with the following attributes:

flag
A *number* interpreted as a flags mask to pass to the element method `:is_selected`. It should not be confused with the flags passed as argument to methods `:select` and `:deselect`, as both flags can be used together but with different meanings!

pattern
A *string* interpreted as a pattern to match the element name using `string.match` from the standard library, see Lua 5.2 §6.4 for details.

class
An *element* interpreted as a *class* to pass to the element method `:is_instansceOf`.

list
An *iterable* interpreted as a *list* used to build a *set* and select the elements by their name, i.e. the built predicate will use `tbl[elm.name]` as a *logical*. If the *iterable* is a single item, e.g. a *string*, it will be converted first to a *list*.

table
A *mappable* interpreted as a *set* used to select the elements by their name, i.e. the built predicate will use `tbl[elm.name]` as a *logical*. If the *mappable* contains a *list* or is a single item, it will be converted first to a *list* and its *set* part will be discarded.

select
A *callable* interpreted as the selector itself, which allows to build any kind of predicate or to complete the restrictions already built above.

subelem
A *boolean* indicating to include or not the sub-elements in the scanning loop. The predicate and the action receive the sub-element and its sub-index as first and second argument, and the main element index as third argument.

All these attributes are used in the aforementioned order to incrementally build predicates that are combined with logical conjunctions, i.e. `and`'ed, to give the final predicate used by the `:foreach` method. If only one of these attributes is needed, it is possible to pass it directly in `sel`, not as an attribute in a *set*, and its type will be used to determine the kind of predicate to build. For example, `self:foreach(act, monitor)` is equivalent to `self:foreach\{action=act, class=monitor}`.

1.7 Indexes, names and counts

Indexing a sequence triggers a complex look up mechanism where the arguments will be interpreted in various ways as described in the `:__index` metamethod. A *number* will be interpreted as a relative slot index in the list of elements, and a negative index will be considered as relative to the end of the sequence, i.e. `-1` is the `$end` marker. Non- *number* will be interpreted first as an object key (can be anything), looking for sequence methods or attributes; then as an element name if nothing was found.

If an element exists but its name is not unique in the sequence, an *iterable* is returned. An *iterable* supports the length `#` operator to retrieve the number of elements with the same name, the indexing operator `[]` waiting for a count n to retrieve the n-th element from the start with that name, and the iterator `ipairs` to use with generic `for` loops.

The returned *iterable* is in practice a proxy, i.e. a fake intermediate object that emulates the expected behavior, and any attempt to access the proxy in another manner should raise a runtime error.

Warning: The indexing operator `[]` interprets a *number* as a (relative) element index as the method `:index`, while the method `:index_of`} interprets a *number* as a (relative) element s-position [m].

The following example shows how to access to the elements through indexing and the *iterable*::

```
local sequence, drift, marker in MAD.element
local seq = sequence {
drift 'df' { id=1 }, marker 'mk' { id=2 },
drift 'df' { id=3 }, marker 'mk' { id=4 },
drift 'df' { id=5 }, marker 'mk' { id=6 },
}
print(seq[ 1].name) -- display: (*\$start*) (start marker)
print(seq[-1].name) -- display: (*\$end*)   (end   marker)

print(#seq.df, seq.df[3].id)                         -- display: 3   5
for _,e in ipairs(seq.df) do io.write(e.id," ") end -- display: 1 3 5
for _,e in ipairs(seq.mk) do io.write(e.id," ") end -- display: 2 4 6

-- print name of drift with id=3 in absolute and relative to id=6.
print(seq:name_of(4))        -- display: df[2]  (2nd df from start)
print(seq:name_of(2, -2))    -- display: df{-3} (3rd df before last mk)
```

The last two lines of code display the name of the same element but mangled with absolute and relative counts.

section{Iterators and ranges}

Ranging a sequence triggers a complex look up mechanism where the arguments will be interpreted in various ways as described in the `:range_of` method, itself based on the methods `:index_of`} and `:index`. The number of elements selected by a sequence range can be computed by the `:length_of`} method, which accepts an extra *number* of turns to consider in the calculation.

The sequence iterators are created by the methods `:iter` and `:siter`, and both are based on the `:range_of` method as mentioned in their descriptions and includes an extra *number* of turns as for the method `:length_of`, and a direction `1` (forward) or `-1` (backward) for the iteration. The `:siter` differs from the `:iter` by its loop, which returns not only the sequence elements but also *implicit* drifts built on-the-fly when a gap $> 10^{-10}$ m is detected between two sequence elements. Such implicit drift have half-integer indexes and make the iterator "continuous" in s-positions.

The method `:foreach` uses the iterator returned by `:iter` with a range as its sole argument to loop over the elements where to apply the predicate before executing the action. The methods `:select`, `:deselect`, `:filter`, `:install`,

`:replace`, `:remove`, `:move`, and `:misalign` are all based directly or indirectly on the `:foreach` method. Hence, to iterate backward over a sequence range, these methods have to use either its *list* form or a numerical range. For example the invocation `seq:foreach(\e -> print(e.name), {2, 2, 'idx', -1)` will iterate backward over the entire sequence `seq` excluding the `$start` and `$end` markers, while the invocation `seq:foreach(\e -> print(e.name), 5..2..-1)` will iterate backward over the elements with s-positions sitting in the interval $[2, 5]$ m.

The tracking commands `survey` and `track` use the iterator returned by `:siter` for their main loop, with their `range`, `nturn` and `dir` attributes as arguments. These commands also save the iterator states in their `mflw` to allow the users to run them `nstep` by `nstep`, see commands *survey* and *track* for details.

The following example shows how to access to the elements with the `:foreach` method::

```
local sequence, drift, marker in MAD.element
local observed in MAD.element.flags
local seq = sequence {
drift 'df' { id=1 }, marker 'mk' { id=2 },
drift 'df' { id=3 }, marker 'mk' { id=4 },
drift 'df' { id=5 }, marker 'mk' { id=6 },
}

local act = \e -> print(e.name,e.id)
seq:foreach(act, "df[2]/mk[3]")
-- display:
df   3
mk   4
df   5
mk   6

seq:foreach{action=act, range="df[2]/mk[3]", class=marker}
-- display: markers at ids 4 and 6
seq:foreach{action=act, pattern=(*\verb+"^[^$]"+*)}
-- display: all elements except (*\verb+$start and $end+*) markers
seq:foreach{action=\e -> e:select(observed), pattern="mk"}
-- same as: seq:select(observed, {pattern="mk"})

local act = \e -> print(e.name, e.id, e:is_observed())
seq:foreach{action=act, range=(*\verb+"#s/#e"+*)}
-- display:
(*\$start*)   nil  false
df       1    false
mk       2    true
df       3    false
mk       4    true
df       5    false
mk       6    true
(*\$end*)     nil  true
```

1.8 Examples

1.8.1 FODO cell

```
local sequence, sbend, quadrupole, sextupole, hkicker, vkicker, marker in MAD.element
local mkf = marker 'mkf' {}
local ang=2*math.pi/80
local fodo = sequence 'fodo' { refer='entry',
mkf             { at=0, shared=true      }, -- mark the start of the fodo
quadrupole 'qf' { at=0, l=1  , k1=0.3    },
sextupole  'sf' {       l=0.3, k2=0      },
hkicker    'hk' {       l=0.2, kick=0    },
sbend      'mb' { at=2, l=2  , angle=ang },

quadrupole 'qd' { at=5, l=1  , k1=-0.3   },
sextupole  'sd' {       l=0.3, k2=0      },
vkicker    'vk' {       l=0.2, kick=0    },
sbend      'mb' { at=7, l=2  , angle=ang },
}
local arc = sequence 'arc' { refer='entry', 10*fodo }
fodo:dumpseq() ; print(fodo.mkf, mkf)
```

Display:

```
sequence: fodo, l=9
idx  kind          name          l          dl       spos       upos    uds
001  marker        $start  0.000       0        0.000      0.000   0.000
002  marker        mkf     0.000       0        0.000      0.000   0.000
003  quadrupole    qf      1.000       0        0.000      0.000   0.000
004  sextupole     sf      0.300       0        1.000      1.000   0.000
005  hkicker       hk      0.200       0        1.300      1.300   0.000
006  sbend         mb      2.000       0        2.000      2.000   0.000
007  quadrupole    qd      1.000       0        5.000      5.000   0.000
008  sextupole     sd      0.300       0        6.000      6.000   0.000
009  vkicker       vk      0.200       0        6.300      6.300   0.000
010  sbend         mb      2.000       0        7.000      7.000   0.000
011  marker        $end    0.000       0        9.000      9.000   0.000
marker : 'mkf' 0x01015310e8  marker: 'mkf' 0x01015310e8 -- same marker
```

1.8.2 SPS compact description

The following dummy example shows a compact definition of the SPS mixing elements, beam lines and sequence definitions. The elements are zero-length, so the lattice is too.

```
local drift, sbend, quadrupole, bline, sequence in MAD.element

-- elements (empty!)
local ds = drift      'ds' {}
local dl = drift      'dl' {}
local dm = drift      'dm' {}
local b1 = sbend      'b1' {}
```

(continues on next page)

(continued from previous page)

```
local b2 = sbend      'b2' {}
local qf = quadrupole 'qf' {}
local qd = quadrupole 'qd' {}

-- subsequences
local pf  = bline 'pf'  {qf,2*b1,2*b2,ds}           -- #: 6
local pd  = bline 'pd'  {qd,2*b2,2*b1,ds}           -- #: 6
local p24 = bline 'p24' {qf,dm,2*b2,ds,pd}          -- #: 11 (5+6)
local p42 = bline 'p42' {pf,qd,2*b2,dm,ds}          -- #: 11 (6+5)
local p00 = bline 'p00' {qf,dl,qd,dl}               -- #: 4
local p44 = bline 'p44' {pf,pd}                     -- #: 12 (6+6)
local insert = bline 'insert' {p24,2*p00,p42}       -- #: 30 (11+2*4+11)
local super  = bline 'super'  {7*p44,insert,7*p44}  -- #: 198 (7*12+30+7*12)

-- final sequence
local SPS = sequence 'SPS' {6*super}                -- # = 1188 (6*198)

-- check number of elements and length
print(#SPS, SPS.l)  -- display: 1190  0 (no element length provided)
```

1.8.3 Installing elements I

The following example shows how to install elements and subsequences in an empty initial sequence::

```
local sequence, drift in MAD.element
local seq   = sequence "seq" { l=16, refer="entry", owner=true }
local sseq1 = sequence "sseq1" {
at=5, l=6 , refpos="centre", refer="entry",
drift "df1'" {l=1, at=-4, from="end"},
drift "df2'" {l=1, at=-2, from="end"},
drift "df3'" {     at= 5            },
}
local sseq2 = sequence "sseq2" {
at=14, l=6, refpos="exit", refer="entry",
drift "df1''" { l=1, at=-4, from="end"},
drift "df2''" { l=1, at=-2, from="end"},
drift "df3''" {      at= 5            },
}
seq:install {
drift "df1" {l=1, at=1},
sseq1, sseq2,
drift "df2" {l=1, at=15},
} :dumpseq()
```

Display:

```
sequence: seq, l=16
idx  kind         name      l          dl       spos       upos     uds
001  marker       $start*   0.000       0       0.000      0.000    0.000
002  drift        df1       1.000       0       1.000      1.000    0.000
003  drift        df1'      1.000       0       4.000      4.000    0.000
```

(continues on next page)

(continued from previous page)

```
004  drift         df2'       1.000       0       6.000      6.000    0.000
005  drift         df3'       0.000       0       7.000      7.000    0.000
006  drift         df1''      1.000       0      10.000     10.000    0.000
007  drift         df2''      1.000       0      12.000     12.000    0.000
008  drift         df3''      0.000       0      13.000     13.000    0.000
009  drift         df2        1.000       0      15.000     15.000    0.000
010  marker        $end       0.000       0      16.000     16.000    0.000
```

1.8.4 Installing elements II

The following more complex example shows how to install elements and subsequences in a sequence using a selection and the packed form for arguments::

```
local mk   = marker   "mk"   { }
local seq  = sequence "seq" { l = 10, refer="entry",
mk "mk1" { at = 2 },
mk "mk2" { at = 4 },
mk "mk3" { at = 8 },
}
local sseq = sequence "sseq" { l = 3 , at = 5, refer="entry",
drift "df1'" { l = 1, at = 0 },
drift "df2'" { l = 1, at = 1 },
drift "df3'" { l = 1, at = 2 },
}
seq:install {
class    = mk,
elements = {
   drift "df1" { l = 0.1, at = 0.1, from="selected" },
   drift "df2" { l = 0.1, at = 0.2, from="selected" },
   drift "df3" { l = 0.1, at = 0.3, from="selected" },
   sseq,
   drift "df4" { l = 1, at = 9 },
}
}

seq:dumpseq()
```

```
sequence: seq, l=10
idx  kind          name      l          dl        spos       upos     uds
001  marker        $start    0.000       0        0.000      0.000    0.000
002  marker        mk1       0.000       0        2.000      2.000    0.000
003  drift         df1       0.100       0        2.100      2.100    0.000
004  drift         df2       0.100       0        2.200      2.200    0.000
005  drift         df3       0.100       0        2.300      2.300    0.000
006  marker        mk2       0.000       0        4.000      4.000    0.000
007  drift         df1       0.100       0        4.100      4.100    0.000
008  drift         df2       0.100       0        4.200      4.200    0.000
009  drift         df3       0.100       0        4.300      4.300    0.000
010  drift         df1'      1.000       0        5.000      5.000    0.000
011  drift         df2'      1.000       0        6.000      6.000    0.000
```

(continues on next page)

(continued from previous page)

```
012  drift        df3'     1.000      0      7.000      7.000   0.000
013  marker       mk3      0.000      0      8.000      8.000   0.000
014  drift        df1      0.100      0      8.100      8.100   0.000
015  drift        df2      0.100      0      8.200      8.200   0.000
016  drift        df3      0.100      0      8.300      8.300   0.000
017  drift        df4      1.000      0      9.000      9.000   0.000
018  marker       $end     0.000      0     10.000     10.000   0.000
```

1.9 Random Maths

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t}$$

CHAPTER

TWO

TYPES

Just here to be a link

CHAPTER THREE

GENERIC UTILITIES

Just here to be linked to

CHAPTER

FOUR

ELEMENTARY CONSTANTS

This chapter describes basic mathematical and physiscal constants provided by the module `constant`.

4.1 Mathematical Constants

This section describes basic mathematical constants uniquely defined as macros in the C header `mad_cst.h` and available from C and MAD modules. If these mathematical constants are already provided by the system libraries, they are used instead of the local definitions.

MAD constants	C macros	C constants	Values
eps	DBL_EPSILON	mad_cst_EPS	Smallest representable increment near one
tiny	DBL_MIN	mad_cst_TINY	Smallest representable number
huge	DBL_MAX	mad_cst_HUGE	Largest representable number
inf	INFINITY	mad_cst_INF	Positive infinity, $1/0$
nan	•	•	Canonical NaN, $0/0$
e	M_E	mad_cst_E	$e, \exp(1)$
log2e	M_LOG2E	mad_cst_LOG2E	$\log_2(e)$
log10e	M_LOG10E	mad_cst_LOG10E	$\log_{10}(e)$
ln2	M_LN2	mad_cst_LN2	$\ln(2)$
ln10	M_LN10	mad_cst_LN10	$\ln(10)$
lnpi	M_LNPI	mad_cst_LNPI	$\ln(\pi)$
pi	M_PI	mad_cst_PI	π
twopi	M_2PI	mad_cst_2PI	2π
pi_2	M_PI_2	mad_cst_PI_2	$\pi/2$
pi_4	M_PI_4	mad_cst_PI_4	$\pi/4$
one_pi	M_1_PI	mad_cst_1_PI	$1/\pi$
two_pi	M_2_PI	mad_cst_2_PI	$2/\pi$
sqrt2	M_SQRT2	mad_cst_SQRT2	$\sqrt{2}$
sqrt3	M_SQRT3	mad_cst_SQRT3	$\sqrt{3}$
sqrtpi	M_SQRTPI	mad_cst_SQRTPI	$\sqrt{\pi}$
sqrt1_2	M_SQRT1_2	mad_cst_SQRT1_2	$\sqrt{1/2}$
sqrt1_3	M_SQRT1_3	mad_cst_SQRT1_3	$\sqrt{1/3}$
one_sqrtpi	M_1_SQRTPI	mad_cst_1_SQRTPI	$1/\sqrt{\pi}$
two_sqrtpi	M_2_SQRTPI	mad_cst_2_SQRTPI	$2/\sqrt{\pi}$
raddeg	M_RADDEG	mad_cst_RADDEG	$180/\pi$
degrad	M_DEGRAD	mad_cst_DEGRAD	$\pi/180$

4.2 Physical Constants

This section describes basic physical constants uniquely defined as macros in the C header `mad_cst.h` and available from C and MAD modules.

MAD constants	C macros	C constants	Values
minlen	P_MINLEN	mad_cst_MINLEN	Minimum length tolerance, 10^-10 in **[m]**
minang	P_MINANG	mad_cst_MINANG	Minimum angle tolerance, 10^-10 in **[m^{-1}]**
minstr	P_MINSTR	mad_cst_MINSTR	Minimum strength tolerance, 10^-10 in **[rad]**

The following table lists some physical constants from the CODATA 2018 sheet.

MAD constants	C macros	C constants	Values
`clight`	`P_CLIGHT`	`mad_cst_CLIGHT`	Speed of light, c in **[m/s]**
`mu0`	`P_MU0`	`mad_cst_MU0`	Permeability of vacuum, μ_0 in **[T.m/A]**
`epsilon0`	`P_EPSILON0`	`mad_cst_EPSILON0`	Permittivity of vacuum, ϵ_0 in **[F/m]**
`qelect`	`P_QELECT`	`mad_cst_QELECT`	Elementary electric charge, e in **[C]**
`hbar`	`P_HBAR`	`mad_cst_HBAR`	Reduced Plack's constant, $\hbar$ in **[GeV.s]**
`amass`	`P_AMASS`	`mad_cst_AMASS`	Unified atomic mass, m_uc^2 in **[GeV]**
`emass`	`P_EMASS`	`mad_cst_EMASS`	Electron mass, m_ec^2 in **[GeV]**
`pmass`	`P_PMASS`	`mad_cst_PMASS`	Proton mass, m_pc^2 in **[GeV]**
`nmass`	`P_NMASS`	`mad_cst_NMASS`	Neutron mass, m_nc^2 in **[GeV]**
`mumass`	`P_MUMASS`	`mad_cst_MUMASS`	Muon mass, $m_\mu c^2$ in **[GeV]**
`deumass`	`P_DEUMASS`	`mad_cst_DEUMASS`	Deuteron mass, m_dc^2 in **[GeV]**
`eradius`	`P_ERADIUS`	`mad_cst_ERADIUS`	Classical electron radius, r_e in **[m]**
`alphaem`	`P_ALPHAEM`	`mad_cst_ALPHAEM`	Fine-structure constant, α

CHAPTER

FIVE

ELEMENTS

Just a link

5.1 Misalignment

CHAPTER

SIX

TRACK

Just a link

CHAPTER SEVEN

SURVEY

Just a link

CHAPTER

EIGHT

ELEMENTARY FUNCTIONS

This chapter describes elementary functions provided by the module `gmath`. This module extends the standard module `math` with *generic* functions working on any type that implements the methods with the same name. For example, the code `gmath.sin(a)` will call `math.sin(a)` if `a` is a *number*, otherwise it will calling the method `a:sin()`, i.e. delegate the call to `a`. This is how MAD-NG handles few types like *numbers*, *complex* number, *matrix* and *TPSA* within a single code.

8.1 Generic Operators

Generic operators are named functions that rely on associated operators, which themselves can be redefined by their associated metamethods.

Operators	Return values	Metamethods
`unm(x)`	`-x`	__unm(x,_)
`add(x,y)`	`x + y`	__add(x,y)
`sub(x,y)`	`x - y`	__sub(x,y)
`mul(x,y)`	`x * y`	__mul(x,y)
`div(x,y)`	`x / y`	__div(x,y)
`mod(x,y)`	`x % y`	__mod(x,y)
`pow(x,y)`	`x ^ y`	__pow(x,y)
`sqr(x)`	`x * x`	•
`inv(x)`	`1 / x`	•
`emul(x,y,r_)`	`x .* y`	__emul(x,y,r_)
`ediv(x,y,r_)`	`x ./ y`	__ediv(x,y,r_)
`emod(x,y,r_)`	`x .% y`	__emod(x,y,r_)
`epow(x,y,r_)`	`x .^ y`	__epow(x,y,r_)

8.2 Generic Functions (real-like)

Real-like generic functions forward the call to the method of the same name from the first argument when the later is not a *number*.

Functions	Return values	C functions
`abs (x)`	$\|x\|$	
`acos (x)`	$\cos^{-1}(x)$	
`acosh (x)`	$\cosh^{-1}(x)$	`acosh()`
`acot (x)`	$\cot^{-1}(x)$	
`acoth (x)`	$\coth^{-1}(x)$	`atanh()`
`asin (x)`	$\sin^{-1}(x)$	
`asinc (x)`	$\frac{\sin^{-1}(x)}{x}$	
`asinh (x)`	$\sinh^{-1}(x)$	`asinh()`
`asinhc (x)`	$\frac{\sinh^{-1}(x)}{x}$	
`atan (x)`	$\tan^{-1}(x)$	
`atan2 (x,y)`	$\tan^{-1}(\frac{x}{y})$	
`atanh (x)`	$\tanh^{-1}(x)$	`atanh()`
`ceil (x)`	$\mathrm{ceil}(x)$	
`cos (x)`	$\cos(x)$	
`cosh (x)`	$\cosh(x)$	
`cot (x)`	$\cot(x)$	
`coth (x)`	$\coth(x)$	
`deg2rad(x)`	$\frac{\pi}{180}x$	
`exp (x)`	$\exp(x)$	
`floor (x)`	$\mathrm{floor}(x)$	
`frac (x)`	$\mathrm{frac}(x)$	
`hypot (x,y)`	$\sqrt{x^2+y^2}$	`hypot()`
`hypot3 (x,y,z)`	$\sqrt{x^2+y^2+z^2}$	`hypot()`
`invsqrt(x,v_)`	$\frac{v}{\sqrt{x}}$	
`log (x)`	$\log(x)$	
`log10 (x)`	$\log 10(x)$	
`pow (x,y)`	x^y	
`rad2deg(x)`	$\frac{180}{pi}x$	
`round (x)`	$\mathrm{round}(x)$	`round()`
`sign (x)`	$-1, 0$ or 1	`mad_num_sign()`
`sign1 (x)`	-1 or 1	`mad_num_sign1()`
`sin (x)`	$\sin(x)$	
`sinc (x)`	$\frac{\sin(x)}{x}$	
`sinh (x)`	$\sinh(x)$	
`sinhc (x)`	$\frac{\sinh(x)}{x}$	
`sqrt (x)`	$\sqrt{x}$	
`tan (x)`	$\tan(x)$	
`tanh (x)`	$\tanh(x)$	
`lgamma (x,tol)`	$\ln\|\Gamma(x)\|$	`lgamma()`
`tgamma (x,tol)`	$\Gamma(x)$	`tgamma()`
`trunc (x)`	$\mathrm{trunc}(x)$	
`unit (x)`	$\frac{x}{\|x\|}$	

8.3 Generic Functions (complex-like)

Complex-like generic functions forward the call to the method of the same name from the first argument when the later is not a *number*, otherwise it implements a real-like compatibility layer using the equivalent representation $x + 0i$.

Functions	Return values
`cabs (z)`	$\|z\|$
`carg (z)`	$\arg(z)$
`conj (z)`	z^*
`cplx (x,y)`	$x + i\,y$
`imag (z)`	$\Im(z)$
`polar(z)`	$\|z\|\,e^{i\arg(z)}$
`proj (z)`	$\operatorname{Proj}(z)$
`real (z)`	$\Re(z)$
`rect (z)`	$\Re(z)\cos(\Im(z)) + i\,\Re(z)\sin(\Im(z))$
`reim (z)`	$(\Re(z), \Im(z))$

8.4 Generic Functions (Error-like)

Error-like generic functions forward the call to the method of the same name from the first argument when the later is not a *number*, otherwise it calls a C wrapper to corresponding function from the Faddeeva library from the MIT (see `mad_num.c`).

Functions	C functions for reals	C functions for complex
`erf (x,tol)`	`mad_num_erf`	`mad_cnum_erf`
`erfc (x,tol)`	`mad_num_erfc`	`mad_cnum_erfc`
`erfi (x,tol)`	`mad_num_erfi`	`mad_cnum_erfi`
`erfcx(x,tol)`	`mad_num_erfcx`	`mad_cnum_erfcx`
`wf (x,tol)`	`mad_num_wf`	`mad_cnum_wf`

8.5 Generic Functions (Length-Angle based)

Length-Angle base generic function relies on the following elementary relations between length and angle.

$$l_{\text{arc}} = ar = \frac{l_{\text{cord}}}{\operatorname{sinc}(\frac{a}{2})} l_{\text{cord}} = 2r\sin(\frac{a}{2}) = l_{\text{arc}}\operatorname{sinc}(\frac{a}{2})$$

Functions	Return values
`arc2cord(l,a)`	$l\operatorname{sinc}(\frac{a}{2})$
`arc2len (l,a)`	$l\operatorname{sinc}(\frac{a}{2})cos(a)$
`cord2arc(l,a)`	$\frac{l}{\operatorname{sinc}(\frac{a}{2})}$
`cord2len(l,a)`	$lcos(a)$
`len2arc (l,a)`	$\frac{l}{\operatorname{sinc}(\frac{a}{2})cos(a)}$
`len2cord(l,a)`	$\frac{l}{cos(a)}$
`rangle (a,r)`	$a + 2\pi\operatorname{round}(\frac{r-a}{2\pi})$

8.6 Generic Functions (Folding-Left based)

Functions	Return values
`sumsqr (x,y)`	$x^2 + y^2$
`sumabs (x,y)`	$\lvert x\rvert + \lvert y\rvert$
`minabs (x,y)`	$\min(\lvert x\rvert, \lvert y\rvert)$
`maxabs (x,y)`	$\max(\lvert x\rvert, \lvert y\rvert)$
`sumysqr(x,y)`	$x + y^2$
`sumyabs(x,y)`	$x + \lvert y\rvert$
`minyabs(x,y)`	$\min(x, \lvert y\rvert)$
`maxyabs(x,y)`	$\max(x, \lvert y\rvert)$

8.7 Non-Generic Functions

Functions	C or math functions
`deg`	`math.deg`
`fact`	`mad_num_fact`, $n!$
`fmod`	`math.fmod`
`frexp`	`math.frexp`
`invfact`	`mad_num_invfact`, $1/n!$
`ldexp`	`math.ldexp`
`max`	`math.max`
`min`	`math.min`
`modf`	`math.modf`
`rad`	`math.rad`

8.8 Random number generators

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX