MAD-NG Reference Manual

Laurent Deniau
Accelerator Beam Physics,
CERN, Meyrin, Switzerland.

Abstract

The Methodical Accelerator Design – Next Generation application is an all-in-one standalone versatile tool for particle accelerator design, modeling, and optimization, and for beam dynamics and optics studies. Its general purpose scripting language is based on the simple yet powerful Lua programming language (with a few extensions) and embeds the state-of-art Just-In-Time compiler LuaJIT. Its physics is based on symplectic integration of differential maps made out of GTPSA (Generalized Truncated Power Series). The physics of the transport maps and the normal form analysis were both strongly inspired by the PTC/FPP library from E. Forest. MAD-NG development started in 2016 by the author as a side project of MAD-X, hence MAD-X users should quickly become familiar with its ecosystem, e.g. lattices definition.

`http://cern.ch/mad`

Keywords

Methodical Accelerator Design; Accelerator beam physics; Scientific computing; JIT compiler; C and Lua programming.

Contents

Part I

LANGUAGE

Chapter 1. Introduction

1 Presentation

The Methodical Accelerator Design – Next Generation application is an all-in-one standalone versatile tool for particle accelerator design, modeling, and optimization, and for beam dynamics and optics studies. Its general purpose scripting language is based on the simple yet powerful Lua programming language (with a few extensions) and embeds the state-of-art Just-In-Time compiler LuaJIT. Its physics is based on symplectic integration of differential maps made out of GTPSA (Generalized Truncated Power Series). The physics of the transport maps and the normal form analysis were both strongly inspired by the PTC/FPP library from E. Forest. MAD-NG development started in 2016 by the author as a side project of MAD-X, hence MAD-X users should quickly become familiar with its ecosystem, e.g. lattices definition.

MAD-NG is free open-source software, distributed under the GNU General Public License v3.[1] The source code, units tests[2], integration tests, and examples are all available on its Github repository, including the documentation and its LaTeX source. For convenience, the binaries and few examples are also made available from the releases repository located on the AFS shared file system at CERN.

2 Installation

Download the binary corresponding to your platform from the releases repository and install it in a local directory. Update (or check) that the `PATH` environment variable contains the path to your local directory or prefix `mad` with this path to run it. Rename the application from `mad-arch-v.m.n` to `mad` and make it executable with the command '`chmod u+x mad`' on Unix systems or add the `.exe` extension on Windows.

```
$ ./mad - h
usage: ./mad [options]... [script [args]...].
Available options are:
        - e chunk       Execute string 'chunk'.
        - l name        Require library 'name'.
        - b ...         Save or list bytecode.
        - j cmd         Perform JIT control command.
        - O[opt]        Control JIT optimizations.
        - i             Enter interactive mode after executing 'script'.
        - q             Do not show version information.
        - M             Do not load MAD environment.
        - Mt[=num]      Set initial MAD trace level to 'num'.
        - MT[=num]      Set initial MAD trace level to 'num' and location.
        - E             Ignore environment variables.
        --               Stop handling options.
        -                Execute stdin and stop handling options.
```

[1] MAD-NG embeds the libraries FFTW NFFT and NLopt released under GNU (L)GPL too.
[2] MAD-NG has few thousands unit tests that do few millions checks, and it is constantly growing.

2.1 Releases version

MAD-NG releases are tagged on the Github repository and use mangled binary names on the releases repository, i.e. `mad-arch-v.m.n` where:

arch
: is the platform architecture for binaries among `linux`, `macos` and `windows`.

v
: is the **v**ersion number, `0` meaning beta-version under active development.

m
: is the **m**ajor release number corresponding to features completeness.

n
: is the mi**n**or release number corresponding to bug fixes.

3 Interactive Mode

To run MAD-NG in interactive mode, just typewrite its name on the Shell invite like any command-line tool. It is recommended to wrap MAD-NG with the readline wrapper `rlwrap` in interactive mode for easier use and commands history:

```
$ rlwrap ./mad
   ____  __   ______     ______     |   Methodical Accelerator Design
    /  \/  \   /  _  \   /  _  \    |   release: 0.9.0 (OSX 64)
   /  __   /  /  /_/ /  /  /_/ /   |   support: http://cern.ch/mad
  /__/  /_/  /__/ /_/  /_____ /     |   licence: GPL3 (C) CERN 2016+
                                    |   started: 2020-08-01 20:13:51
> print "hello world!"
hello world!"
```

Here the application is assumed to be installed in the current directory '.' and the character '>' is the prompt waiting for user input in interactive mode. If you write an incomplete statement, the interpreter waits for its completion by issuing a different prompt:

```
> print                -- 1st level prompt, incomplete statement
>> "hello world!"       -- 2nd level prompt, complete the statement
hello world!            -- execute
```

Typing the character '=' right after the 1st level prompt is equivalent to call the `print` function:

```
> = "hello world!"      -- 1st level prompt followed by =
hello world!            -- execute print "hello world!"
> = MAD.option.numfmt
% -.10g
```

To quit the application typewrite `Crtl+D` to send `EOF` (end-of-file) on the input,[3] or `Crtl+\` to send the

[3] Note that sending `Crtl+D` twice from MAD-NG invite will quit both MAD-NG and its parent Shell...

SIGQUIT (quit) signal, or Crtl+C to send the stronger SIGINT (interrupt) signal. If the application is stalled or looping for ever, typewriting a single Crtl+\ or Crtl+C twice will stop it:

```
> while true do end    -- loop forever, 1st Crtl+C doesn't stop it
pending interruption in VM! (next will exit)         -- 2nd Crtl+C
interrupted!           -- application stopped

> while true do end    -- loop forever, a single Crtl+\ does stop it
Quit: 3                -- Signal 3 caught, application stopped
```

In interactive mode, each line input is run in its own *chunk*[4], which also rules variables scopes. Hence local, variables are not visible between chunks, i.e. input lines. The simple solutions are either to use global variables or to enclose local statements into the same chunk delimited by the do ... end keywords:

```
> local a = "hello"
> print(a.." world!")
  stdin:1: attempt to concatenate global 'a' (a nil value)
  stack traceback:
  stdin:1: in main chunk
  [C]: at 0x01000325c0

> do                     -- 1st level prompt, open the chunck
>> local a = "hello"     -- 2nd level prompt, waiting for statement completion
>> print(a.." world!")   -- same chunk, local 'a' is visible
>> end                   -- close and execute the chunk
hello world!
> print(a)               -- here 'a' is an unset global variable
nil
> a = "hello"            -- set global 'a'
> print(a.." world!")    -- works but pollutes the global environment
hello world!
```

4 Batch Mode

To run MAD-NG in batch mode, just run it in the shell with files as arguments on the command line:

```
$ ./mad [mad options] myscript1.mad myscript2.mad ...
```

where the scripts contains programs written in the MAD-NG programming language (see *Scripting*).

[4] A chunk is the unit of execution in Lua (see Lua 5.2 §3.3.2).

5 Online Help

MAD-NG is equipped with an online help system[5] useful in interactive mode to quickly search for information displayed in the man-like Unix format :

```
    > help()
Related topics:
MADX, aperture, beam, cmatrix, cofind, command, complex, constant, correct,
ctpsa, cvector, dynmap, element, filesys, geomap, gfunc, gmath, gphys, gplot,
gutil, hook, lfun, linspace, logrange, logspace, match, matrix, mflow,
monomial, mtable, nlogrange, nrange, object, operator, plot, range, reflect,
regex, sequence, strict, survey, symint, symintc, tostring, totable, tpsa,
track, twiss, typeid, utest, utility, vector.

> help "MADX"
NAME
MADX environment to emulate MAD-X workspace.

SYNOPSIS
local lhcb1 in MADX

DESCRIPTION
This module provide the function 'load' that read MADX sequence and optics
files and load them in the MADX global variable. If it does not exist, it will
create the global MADX variable as an object and load into it all elements,
constants, and math functions compatible with MADX.

RETURN VALUES
The MADX global variable.

EXAMPLES
MADX:open()
-- inline definition
MADX:close()

SEE ALSO
element, object.
```

Complementary to the help function, the function show displays the type and value of variables, and if they have attributes, the list of their names in the lexicographic order:

```
> show "hello world!"
:string: hello world!
> show(MAD.option)
:table: MAD.option
```

(continues on next page)

[5] The online help is far incomplete and will be completed, updated and revised as the application evolves.

(continued from previous page)

```
colwidth            :number: 18
hdrwidth            :number: 18
intfmt              :string: % -10d
madxenv             :boolean: false
nocharge            :boolean: false
numfmt              :string: % -.10g
ptcmodel            :boolean: false
strfmt              :string: % -25s
```

Chapter 2. Scripting

The choice of the scripting language for MAD-NG was sixfold: the *simplicity* and the *completeness* of the programming language, the *portability* and the *efficiency* of the implementation, and its easiness to be *extended* and *embedded* in an application. In practice, very few programming languages and implementations fulfill these requirements, and Lua and his Just-In-Time (JIT) compiler LuaJIT were not only the best solutions but almost the only ones available when the development of MAD-NG started in 2016.

1 Lua and LuaJIT

The easiest way to shortly describe these choices is to cite their authors.

"Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping."[1]

"LuaJIT is widely considered to be one of the fastest dynamic language implementations. It has outperformed other dynamic languages on many cross-language benchmarks since its first release in 2005 — often by a substantial margin — and breaks into the performance range traditionally reserved for offline, static language compilers."[2]

Lua and LuaJIT are free open-source software, distributed under the very liberal MIT license.

MAD-NG embeds a patched version of LuaJIT 2.1, a very efficient implementation of Lua 5.2.[3] Hence, the scripting language of MAD-NG is Lua 5.2 with some extensions detailed in the next section, and used for both, the development of most parts of the application, and as the user scripting language. There is no strong frontier between these two aspects of the application, giving full access and high flexibility to the experienced users. The filename extension of MAD-NG scripts is `.mad`.

Learning Lua is easy and can be achieved within a few hours. The following links should help to quickly become familiar with Lua and LuaJIT:

- Lua website.
- Lua 5.2 manual for MAD-NG (30 p. PDF).
- Lua 5.0 free online book (old).
- LuaJIT website.
- LuaJIT wiki.
- LuaJIT 2.1 documentation.
- LuaJIT 2.1 on GitHub.

[1] This text is taken from the "What is Lua?" section of the Lua website.
[2] This text is taken from the "Overview" section of the LuaJIT website.
[3] The `ENV` feature of Lua 5.2 is not supported and will never be according to M. Pall.

2 Lua primer

The next subsections introduce the basics of the Lua programming language with syntax highlights, namely variables, control flow, functions, tables and methods.[4]

2.1 Variables

```lua
n = 42  -- All numbers are doubles, but the JIT may specialize them.
-- IEEE-754 64-bit doubles have 52 bits for storing exact int values;
-- machine precision is not a problem for ints < 1e16.

s = 'walternate'  -- Immutable strings like Python.
t = "double-quotes are also fine"
u = [[ Double brackets
       start and end
       multi-line strings.]]
v = "double-quotes \z

     are also fine" -- \z eats next whitespaces
t, u, v = nil  -- Undefines t, u, v.
-- Lua has multiple assignments and nil completion.
-- Lua has garbage collection.

-- Undefined variables return nil. This is not an error:
foo = anUnknownVariable  -- Now foo = nil.
```

2.2 Control flow

```lua
-- Blocks are denoted with keywords like do/end:
while n < 50 do
  n = n + 1  -- No ++ or += type operators.
end

-- If clauses:
if n > 40 then
  print('over 40')
elseif s ~= 'walternate' then  -- ~= is not equals.
  -- Equality check is == like Python; ok for strs.
  io.write('not over 40\n')  -- Defaults to stdout.
else
  -- Variables are global by default.
  thisIsGlobal = 5  -- Camel case is common.
```

(continues on next page)

[4] This primer was adapted from the blog "Learn Lua in 15 minutes" by T. Neylon.

(continued from previous page)

```
  -- How to make a variable local:
  local line = io.read()  -- Reads next stdin line.
  -- String concatenation uses the .. operator:
  print('Winter is coming, '..line)
end

-- Only nil and false are falsy; 0 and '' are true!
aBoolValue = false
if not aBoolValue then print('was false') end

-- 'or' and 'and' are short-circuited.
-- This is similar to the a?b:c operator in C/js:
ans = aBoolValue and 'yes' or 'no'  --> ans = 'no'

-- numerical for begin, end[, step] (end included)
revSum = 0
for j = 100, 1, -1 do revSum = revSum + j end
```

2.3 Functions

```
function fib(n)
  if n < 2 then return 1 end
  return fib(n - 2) + fib(n - 1)
end

-- Closures and anonymous functions are ok:
function adder(x)
  -- The returned function is created when adder is
  -- called, and captures the value of x:
  return function (y) return x + y end
end
a1 = adder(9)
a2 = adder(36)
print(a1(16))  --> 25
print(a2(64))  --> 100

-- Returns, func calls, and assignments all work with lists
-- that may be mismatched in length.
-- Unmatched receivers get nil; unmatched senders are discarded.

x, y, z = 1, 2, 3, 4
-- Now x = 1, y = 2, z = 3, and 4 is thrown away.

function bar(a, b, c)
```

(continues on next page)

(continued from previous page)

```
  print(a, b, c)
  return 4, 8, 15, 16, 23, 42
end

x, y = bar('zaphod')  --> prints "zaphod  nil nil"
-- Now x = 4, y = 8, values 15,..,42 are discarded.

-- Functions are first-class, may be local/global.
-- These are the same:
function f(x) return x * x end
f = function (x) return x * x end

-- And so are these:
local function g(x) return math.sin(x) end
local g; g  = function (x) return math.sin(x) end
-- the 'local g' decl makes g-self-references ok.

-- Calls with one string param don't need parens:
print 'hello'  -- Works fine.
```

2.4 Tables

```
-- Tables = Lua's only compound data structure;
--   they are associative arrays, i.e. hash-lookup dicts;
--   they can be used as lists, i.e. sequence of non-nil values.

-- Dict literals have string keys by default:
t = {key1 = 'value1', key2 = false, ['key.3'] = true }

-- String keys looking as identifier can use dot notation:
print(t.key1, t['key.3']) -- Prints 'value1 true'.
-- print(t.key.3)         -- Error, needs explicit indexing by string
t.newKey = {}             -- Adds a new key/value pair.
t.key2 = nil              -- Removes key2 from the table.

-- Literal notation for any (non-nil) value as key:
u = {['@!#'] = 'qbert', [{}] = 1729, [6.28] = 'tau'}
print(u[6.28])  -- prints "tau"

-- Key matching is basically by value for numbers
-- and strings, but by identity for tables.
a = u['@!#']  -- Now a = 'qbert'.
b = u[{}]     -- We might expect 1729, but it's nil:
```

(continues on next page)

(continued from previous page)

```
-- A one-table-param function call needs no parens:
function h(x) print(x.key1) end
h{key1 = 'Sonmi~451'}  -- Prints 'Sonmi~451'.

for key, val in pairs(u) do -- Table iteration.
  print(key, val)
end

-- List literals implicitly set up int keys:
l = {'value1', 'value2', 1.21, 'gigawatts'}
for i,v in ipairs(l) do  -- List iteration.
  print(i,v,l[i])        -- Indices start at 1 !
end
print("length=", #l)     -- # is defined only for sequence.
-- A 'list' is not a real type, l is just a table
-- with consecutive integer keys, treated as a list,
-- i.e. l = {[1]='value1', [2]='value2', [3]=1.21, [4]='gigawatts'}
-- A 'sequence' is a list with non-nil values.
```

2.5 Methods

```
-- Methods notation:
--   function tblname:fn(...) is the same as
--     function tblname.fn(self, ...) with self being the table.
--   calling tblname:fn(...) is the same as
--     tblname.fn(tblname, ...)       here self becomes the table.
t = { disp=function(s) print(s.msg) end, -- Method 'disp'
      msg="Hello world!" }
t:disp() -- Prints "Hello world!"
function t:setmsg(msg) self.msg=msg end  -- Add a new method 'setmsg'
t:setmsg "Good bye!"
t:disp() -- Prints "Good bye!"
```

3 Extensions

The aim of the extensions patches applied to the embedded LuaJIT in MAD-NG is to extend the Lua syntax in handy directions, like for example to support the deferred expression operator. A serious effort has been put to develop a Domain Specific Language (DSL) embedded in Lua using these extensions and the native language features to mimic as much as possible the syntax of MAD-X in the relevant aspects of the language, like the definition of elements, lattices or commands, and ease the transition of MAD-X users.

Bending and extending a programming language like Lua to embed a DSL is more general and challenging than creating a freestanding DSL like in MAD-X. The former is compatible with the huge codebase written

by the Lua community, while the latter is a highly specialized niche language. The chosen approach attempts to get the best of the two worlds.

3.1 Line comment

The line comment operator `!` is valid in MAD-NG, but does not exists in Lua:[5]

```
local a = 1     ! this remaining part is a comment
local b = 2     -- line comment in Lua
```

3.2 Unary plus

The unary plus operator + is valid in MAD-NG, but does not exists in Lua:[5]

```
local a = +1    -- syntax error in Lua
local b = +a    -- syntax error in Lua
```

3.3 Local in table

The local `in` table syntax provides a convenient way to retrieve values from a *mappable* and avoid error-prone repetitions of attributes names. The syntax is as follows:

```
local sin, cos, tan in math      -- syntax error in Lua
local a, b, c in { a=1, b=2, c=3 }
! a, b, c in { a=1, b=2, c=3 }   -- invalid with global variables
```

which is strictly equivalent to the Lua code:

```
local sin, cos, tan = math.sin, math.cos, math.tan
local tbl = { a=1, b=2, c=3 }
local a, b, c = tbl.a, tbl.b, tbl.c
! local sin, cos, tan = math.cos, math.sin, math.tan   -- nasty typo
```

The JIT has many kinds of optimization to improve a lot the execution speed of the code, and these work much better if variables are declared `local` with minimal lifespan. *This language extension is of first importance for writing fast clean code!*

[5] This feature was introduced to ease the automatic translation of lattices from MAD-X to MAD-NG.

3.4 Lambda function

The lambda function syntax is pure syntactic sugar for function definition and therefore fully compatible with the Lua semantic. The following definitions are all semantically equivalent:

```
local f = function(x) return x^2 end  -- Lua syntax
local f = \x x^2                      -- most compact form
local f = \x -> x^2                   -- most common form
local f = \(x) -> x^2                 -- for readability
local f = \(x) -> (x^2)               -- less compact form
local f = \x (x^2)                    -- uncommon valid form
local f = \(x) x^2                    -- uncommon valid form
local f = \(x) (x^2)                  -- uncommon valid form
```

The important point is that no space must be present between the *lambda* operator \ and the first formal parameter or the first parenthesis; the former will be considered as an empty list of parameters and the latter as an expressions list returning multiple values, and both will trigger a syntax error. For the sake of readability, it is possible without changing the semantic to add extra spaces anywhere in the definition, add an arrow operator ->, or add parentheses around the formal parameter list, whether the list is empty or not.

The following examples show *lambda* functions with multiple formal parameters:

```
local f = function(x,y) return x+y end  -- Lua syntax
local f = \x x+y                        -- most compact form
local f = \x,y -> x+y                   -- most common form
local f = \x, y -> x + y                -- aerial style
```

The lambda function syntax supports multiple return values by enclosing the list of returned expressions within (not optional!) parentheses:

```
local f = function(x,y) return x+y, x-y end  -- Lua syntax
local f = \x,y(x+y,x-y)                      -- most compact form
local f = \x,y -> (x+y,x-y)                  -- most common form
```

Extra surrounding parentheses can also be added to disambiguate false multiple return values syntax:

```
local f = function(x,y) return (x+y)/2 end  -- Lua syntax
local f = \x,y -> ((x+y)/2)     -- disambiguation: single value returned
! local f = \x,y -> (x+y)/2     -- invalid syntax at '/'

local f = function(x,y) return (x+y)*(x-y) end -- Lua syntax
local f = \x,y -> ((x+y)*(x-y)) -- disambiguation: single value returned
! local f = \x,y -> (x+y)*(x-y) -- invalid syntax at '*'
```

It is worth understanding the error message that invalid syntaxes above would report,

```
file:line: attempt to perform arithmetic on a function value.
```

as it is a bit subtle and needs some explanations: the *lambda* is syntactically closed at the end of the returned expression `(x+y)`, and the following operations / or * are considered as being outside the *lambda* definition, that is applied to the freshly created function itself...

Finally, the *lambda* function syntax supports full function syntax (for consistency) using the *fat* arrow operator => in place of the arrow operator:

```
local c = 0
local f = function(x) c=c+1 return x^2 end    -- Lua syntax
local f = \x => c=c+1 return x^2 end          -- most compact form
```

The fat arrow operator requires the `end` keyword to close syntactically the *lambda* function, and the `return` keyword to return values (if any), as in Lua functions definitions.

3.5 Deferred expression

The deferred expression operator `:=` is semantically equivalent to a *lambda* function without argument. It is syntactically valid only inside *table* constructors (see Lua 5.2 §3.4.8):[5]

```
local var = 10
local fun = \-> var
! local fun := var  -- invalid syntax outside table constructors
local tbl = { v1 := var, v2 =\-> var, v3 = var }
print(tbl.v1(), tbl.v2(), tbl.v3, fun()) -- display: 10 10 10 10
var = 20
print(tbl.v1(), tbl.v2(), tbl.v3, fun()) -- display: 20 20 10 20
```

The deferred expressions hereabove have to be explicitly called to retrieve their values, because they are defined in a *table*. It is a feature of the object model making the deferred expressions behaving like values. Still, it is possible to support deferred expressions as values in a raw *table*, i.e. a table without metatable, using the *deferred* function from the *typeid* module:

```
local deferred in MAD.typeid
local var = 10
local tbl = deferred { v1 := var, v2 =\-> var, v3 = var }
print(tbl.v1, tbl.v2, tbl.v3) -- display: 10 10 10
var = 20
print(tbl.v1, tbl.v2, tbl.v3) -- display: 20 20 10
```

3.6 Ranges

The ranges are created from pairs or triplets of concatenated numbers:[6]

```
start..stop..step   -- order is the same as numerical 'for'
start..stop         -- default step is 1
3..4                -- spaces are not needed around concat operator
3..4..0.1           -- floating numbers are handled
4..3..-0.1          -- negative steps are handled
stop..start..-step  -- operator precedence
```

The default value for unspecified `step` is 1. The Lua syntax has been modified to accept concatenation operator without surrounding spaces for convenience.

Ranges are *iterable* and *lengthable* so the following code excerpt is valid:

```
local rng = 3..4..0.1
print(#rng) -- display: 11
for i,v in ipairs(rng) do print(i,v) end
```

More details on ranges can be found in the *Range* module, especially about the *range* and *logrange* constructors that may adjust `step` to ensure precise loops and iterators behaviors with floating-point numbers.

3.7 Lua syntax and extensions

The operator precedence (see Lua 5.2 §3.4.7) is recapped and extended in Table 2.1 with their precedence level (on the left) from lower to higher priority and their associativity (on the right).

Table2.1: Operators precedence with priority and associativity.

1:	or	left
2:	and	left
3:	< > <= >= ~= ==	left
4:	..	right
5:	+ - (binary)	left
6:	* / %	left
7:	not # - + (unary)	left
8:	^	right
9:	. [] () (call)	left

The *string* literals, *table* constructors, and *lambda* definitions can be combined with function calls (see Lua 5.2 §3.4.9) advantageously like in the object model to create objects in a similar way to MAD-X. The following function calls are semantically equivalent by pairs:

[6] This is the only feature of MAD-NG that is incompatible with the semantic of Lua.

```
! with parentheses                    ! without parentheses
func( 'hello world!' )                func  'hello world!'
func( "hello world!" )                func  "hello world!"
func( [[hello world!]] )              func  [[hello world!]]
func( {...fields...} )                func  {...fields...}
func( \x -> x^2 )                     func  \x -> x^2
func( \x,y -> (x+y,x-y) )             func  \x,y -> (x+y,x-y)
```

4 Types

MAD-NG is based on Lua, a dynamically typed programming language that provides the following *basic types* often italicized in this textbook:

nil
: The type of the value `nil`. Uninitialized variables, unset attributes, mismatched arguments, mismatched return values etc, have `nil` values.

boolean
: The type of the values `true` and `false`.

number
: The type of IEEE 754 double precision floating point numbers. They are exact for integers up to $\pm 2^{53}$ ($\approx \pm 10^{16}$). Values like `0`, `1`, `1e3`, `1e-3` are numbers.

string
: The type of character strings. Strings are "internalized" meaning that two strings with the same content compare equal and share the same memory address: `a="hello"; b="hello"; print(a==b) --display: true`.

table
: The type of tables, see Lua 5.2 §3.4.8 for details. In this textbook, the following qualified types are used to distinguish between two kinds of special use of tables:
  - A *list* is a table used as an array, that is a table indexed by a *continuous* sequence of integers starting from `1` where the length operator `#` has defined behavior.[7]
  - A *set* is a table used as a dictionary, that is a table indexed by keys — strings or other types — or a *sparse* sequence of integers where the length operator `#` has undefined behavior.

function
: The type of functions, see Lua 5.2 §3.4.10 for details. In this textbook, the following qualified types are used to distinguish between few kinds of special use of functions:
  - A *lambda* is a function defined with the `\` syntax.
  - A *functor* is an object[8] that behaves like a function.
  - A *method* is a function called with the `:` syntax and its owner as first argument. A *method* defined with the `:` syntax has an implicit first argument named `self`[9]

thread
: The type of coroutines, see Lua 5.2 §2.6 for details.

[7] The Lua community uses the term *sequence* instead of *list*, which is confusing is the context of MAD-NG.
[8] Here the term "object" is used in the Lua sense, not as an object from the object model of MAD-NG.
[9] This *hidden* methods argument is named `self` in Lua and Python, or `this` in Java and C++.

userdata
: The type of raw pointers with memory managed by Lua, and its companion *lightuserdata* with memory managed by the host language, usually C. They are mainly useful for interfacing Lua with its C API, but MAD-NG favors the faster FFI[10] extension of LuaJIT.

cdata
: The type of C data structures that can be defined, created and manipulated directly from Lua as part of the FFIPage 24, 10 extension of LuaJIT. The numeric ranges, the complex numbers, the (complex) matrices, and the (complex) GTPSA are *cdata* fully compatible with the embedded C code that operates them.

This textbook uses also some extra terms in place of types:

value
: An instance of any type.

reference
: A valid memory location storing some *value*.

logical
: A *value* used by control flow, where `nil` ≡ `false` and *anything-else* ≡ `true`.

4.1 Value vs reference

The types *nil*, *boolean* and *number* have a semantic by *value*, meaning that variables, arguments, return values, etc., hold their instances directly. As a consequence, any assignment makes a copy of the *value*, i.e. changing the original value does not change the copy.

The types *string*, *function*, *table*, *thread*, *userdata* and *cdata* have a semantic by *reference*, meaning that variables, arguments, return values, etc., do not store their instances directly but a *reference* to them. As a consequence, any assignment makes a copy of the *reference* and the instance becomes shared, i.e. references have a semantic by *value* but changing the content of the value does change the copy.[11]

The types *string*, *function*[12], *thread*, `cpx` *cdata* and numeric (`log`)`range` *cdata* have a hybrid semantic. In practice these types have a semantic by *reference*, but they behave like types with semantic by *value* because their instances are immutable, and therefore sharing them is safe.

5 Concepts

The concepts are natural extensions of types that concentrate more on behavior of objectsPage 23, 8 than on types. MAD-NG introduces many concepts to validate objects passed as argument before using them. The main concepts used in this textbook are listed below, see the *typeid* module for more concepts:

lengthable
: An object that can be sized using the length operator `#`. Strings, lists, vectors and ranges are examples of *lengthable* objects.

[10] FFI stands for Foreign Function Interface, an acronym well known in high-level languages communities.
[11] References semantic in Lua is similar to pointers semantic in C, see ISO/IEC 9899:1999 §6.2.5.
[12] Local variables and upvalues of functions can be modified using the `debug` module.

indexable
: An object that can be indexed using the square bracket operator `[]`. Tables, vectors and ranges are examples of *indexable* objects.

iterable
: An object that can be iterated with a loop over indexes or a generic `for` with the `ipairs` iterator. Lists, vectors and ranges are examples of *iterable* objects.

mappable
: An object that can be iterated with a loop over keys or a generic `for` with the `pairs` iterator. Sets and objects (from the object model) are examples of *mappable* objects.

callable
: An object that can be called using the call operator `()`. Functions and functors are examples of *callable* objects.

6 Ecosystem

Fig. 2.1 shows a schematic representation of the ecosystem of MAD-NG, which should help the users to understand the relatioship between the different components of the application. The dashed lines are grouping the items (e.g. modules) by topics while the arrows are showing interdependencies between them and the colors their status.

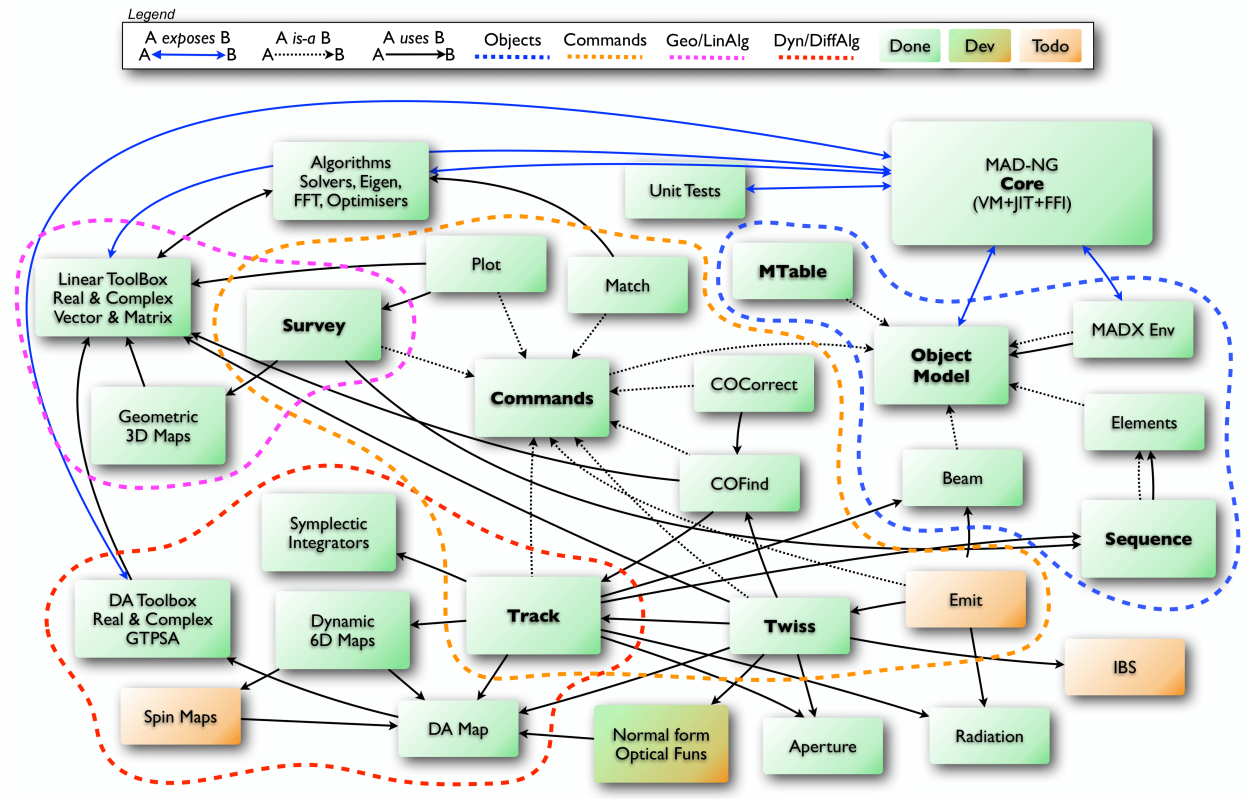

Figure2.1: MAD-NG ecosystem and status.

Chapter 3. Objects

The object model is of key importance as it implements many features used extensively by objects like `beam`, `sequence`, `mtable`, all the commands, all the elements, and the `MADX` environment. The aim of the object model is to extend the scripting language with concepts like objects, inheritance, methods, metamethods, deferred expressions, commands and more.

In computer science, the object model of MAD-NG is said to implement the concepts of prototypical objects, single inheritance and dynamic lookup of attributes:

- A *prototypical object* is an object created from a prototype,[1] named its parent.
- *Single inheritance* specifies that an object has only one direct parent.
- *Dynamic lookup* means that undefined attributes are searched in the parents at *each* read.

A prototype represents the default state and behavior, and new objects can reuse part of the knowledge stored in the prototype by inheritance, or by defining how the new object differs from the prototype. Because any object can be used as a prototype, this approach holds some advantages for representing default knowledge, and incrementally and dynamically modifying them.

1 Creation

The creation of a new object requires to hold a reference to its parent, i.e. the prototype, which indeed will create the child and return it as if it were returned from a function:

```
local object in MAD
local obj = object { }
```

The special *root object* `object` from the `MAD` environment is the parent of *all* objects, including elements, sequences, TFS tables and commands. It provides by inheritance the methods needed to handle objects, environments, and more. In this minimalist example, the created object has `object` as parent, so it is the simplest object that can be created.

It is possible to name immutably an object during its creation:

```
local obj = object 'myobj' { }
print(obj.name) -- display: myobj
```

Here,[2] `obj` is the variable holding the object while the *string* `'myobj'` is the name of the object. It is important to distinguish well the variable that holds the *object* from the object's name that holds the *string*, because they are very often named the same.

It is possible to define attributes during object creation or afterward:

```
local obj = object 'myobj' { a=1, b='hello' }
obj.c = { d=5 } -- add a new attribute c
print(obj.name, obj.a, obj.b, obj.c.d)  -- display: myobj 1 hello 5
```

[1] Objects are not clones of prototypes, they share states and behaviors with their parents but do not hold copies.
[2] This syntax for creating objects eases the lattices translation from MAD-X to MAD-NG.

1.1 Constructors

The previous object creation can be done equivalently using the prototype as a constructor:

```
local obj = object('myobj',{ a=1, b='hello' })
```

An object constructor expects two arguments, an optional *string* for the name, and a required *table* for the attributes placeholder, optionally filled with initial attributes. The table is used to create the object itself, so it cannot be reused to create a different object:

```
local attr = { a=1, b='hello' }
local obj1 = object('obj1',attr) -- ok
local obj2 = object('obj2',attr) -- runtime error, attr is already used.
```

The following objects creations are all semantically equivalent but use different syntax that may help to understand the creation process and avoid runtime errors:

```
-- named objects:
local nobj = object 'myobj'  { }  -- two stages creation.
local nobj = object 'myobj' ({ }) -- idem.
local nobj = object('myobj') { }  -- idem.
local nobj = object('myobj')({ }) -- idem.
local nobj = object('myobj', { }) -- one stage creation.
-- unnamed objects:
local uobj = object   { }         -- one stage creation.
local uobj = object  ({ })        -- idem.
local uobj = object() { }         -- two stages creation.
local uobj = object()({ })        -- idem.
local uobj = object(nil,{ })      -- one stage creation.
```

1.2 Incomplete objects

The following object creation shows how the two stage form can create an incomplete object that can only be used to complete its construction:

```
local obj = object 'myobj'   -- obj is incomplete, table is missing
print(obj.name)              -- runtime error.
obj = obj { }                -- now obj is complete.
print(obj.name)              -- display: myobj
```

Any attempt to use an incomplete object will trigger a runtime error with a message like:

```
file:line: forbidden read access to incomplete object.
```

or

```
file:line: forbidden write access to incomplete object.
```

depending on the kind of access.

1.3 Classes

An object used as a prototype to create new objects becomes a *class*, and a class cannot change, add, remove or override its methods and metamethods. This restriction ensures the behavioral consistency between the children after their creation. An object qualified as *final* cannot create instances and therefore cannot become a class.

1.4 Identification

The `object` module extends the *typeid* module with the `is_object(a)` *function*, which returns `true` if its argument `a` is an object, `false` otherwise:

```
local is_object in MAD.typeid
print(is_object(object), is_object(object{}), is_object{})
-- display: true  true  false
```

It is possible to know the objects qualifiers using the appropriate methods:

```
print(object:is_class(), object:is_final(), object:is_readonly())
-- display: true  false  true
```

1.5 Customizing creation

During the creation process of objects, the metamethod `__init(self)` is invoked if it exists, with the newly created object as its sole argument to let the parent finalize or customize its initialization *before* it is returned. This mechanism is used by commands to run their `:exec()` *method* during their creation.

2 Inheritance

The object model allows to build tree-like inheritance hierarchy by creating objects from classes, themselves created from other classes, and so on until the desired hierarchy is modeled. The example below shows an excerpt of the taxonomy of the elements as implemented by the *element* module, with their corresponding depth levels in comment:

```
local object in MAD                    -- depth level 1
local element = object           {...} -- depth level 2

local drift_element = element    {...} -- depth level 3
```

(continues on next page)

(continued from previous page)

```
local instrument = drift_element {...} -- depth level 4
local monitor  = instrument      {...} -- depth level 5
local hmonitor = monitor         {...} -- depth level 6
local vmonitor = monitor         {...} -- depth level 6

local thick_element = element    {...} -- depth level 3
local tkicker = thick_element    {...} -- depth level 4
local kicker  = tkicker          {...} -- depth level 5
local hkicker = kicker           {...} -- depth level 6
local vicker  = kicker           {...} -- depth level 6
```

2.1 Reading attributes

Reading an attribute not defined in an object triggers a recursive dynamic lookup along the chain of its parents until it is found or the root `object` is reached. Reading an object attribute defined as a *function* automatically evaluates it with the object passed as the sole argument and the returned value is forwarded to the reader as if it were the attribute's value. When the argument is not used by the function, it becomes a *deferred expression* that can be defined directly with the operator `:=` as explained in the section *Deferred expression*. This feature allows to use attributes holding values and functions the same way and postpone design decisions, e.g. switching from simple value to complex calculations without impacting the users side with calling parentheses at every use.

The following example is similar to the second example of the section *Deferred expression*, and it must be clear that `fun` must be explicitly called to retrieve the value despite that its definition is the same as the attribute `v2`.

```
local var = 10
local fun = \-> var -- here := is invalid
local obj = object { v1 := var, v2 =\-> var, v3 = var }
print(obj.v1, obj.v2, obj.v3, fun()) -- display: 10 10 10 10
var = 20
print(obj.v1, obj.v2, obj.v3, fun()) -- display: 20 20 10 20
```

2.2 Writing attributes

Writing to an object uses direct access and does not involve any lookup. Hence setting an attribute with a non-`nil` value in an object hides his definition inherited from the parents, while setting an attribute with `nil` in an object restores the inheritance lookup:

```
local obj1 = object { a=1, b='hello' }
local obj2 = obj1 { a=\s-> s.b..' world' }
print(obj1.a, obj2.a) -- display: 1 hello world
obj2.a = nil
print(obj1.a, obj2.a) -- display: 1 1
```

This property is extensively used by commands to specify their attributes default values or to rely on other commands attributes default values, both being overridable by the users.

It is forbidden to write to a read-only objects or to a read-only attributes. The former can be set using the `:readonly` *method*, while the latter corresponds to attributes with names that start by __, i.e. two underscores.

2.3 Class instances

To determine if an object is an instance of a given class, use the `:is_instanceOf` *method*:

```
local hmonitor, instrument, element in MAD.element
print(hmonitor:is_instanceOf(instrument)) -- display: true
```

To get the list of *public* attributes of an instance, use the `:get_varkeys` *method*:

```
for _,a in ipairs(hmonitor:get_varkeys()) do print(a) end
for _,a in ipairs(hmonitor:get_varkeys(object)) do print(a) end
for _,a in ipairs(hmonitor:get_varkeys(instrument)) do print(a) end
for _,a in ipairs(element:get_varkeys()) do print(a) end
```

The code snippet above lists the names of the attributes set by:

- the object `hmonitor` (only).
- the objects in the hierachy from `hmonitor` to `object` included.
- the objects in the hierachy from `hmonitor` to `instrument` included.
- the object `element` (only), the root of all elements.

2.4 Examples

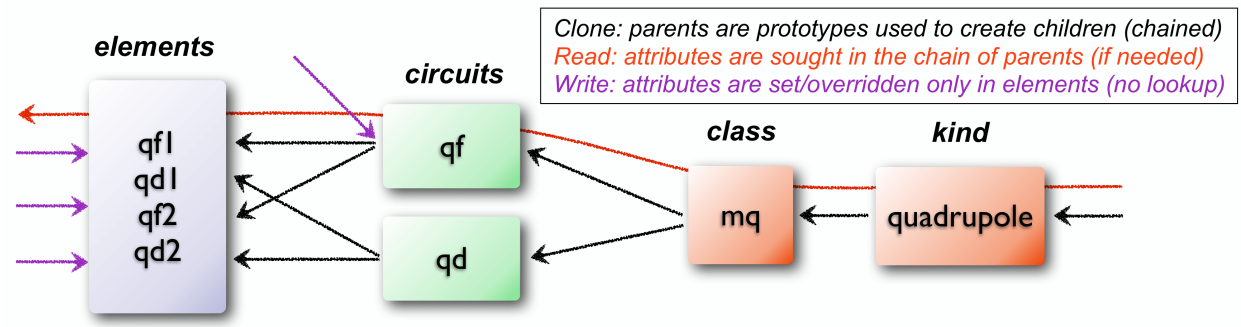

Figure3.1: Object model and inheritance.

Fig. 3.1 summarizes inheritance and attributes lookup with arrows and colors, which are reproduced by the example hereafter:

```
local element, quadrupole in MAD.element    -- kind
local mq  = quadrupole 'mq'  { l  =  2.1  } -- class
local qf  = mq         'qf'  { k1 =  0.05 } -- circuit
local qd  = mq         'qd'  { k1 = -0.06 } -- circuit
local qf1 = qf         'qf1' {}             -- element
... -- more elements
print(qf1.k1)                   -- display: 0.05 (lookup)
qf.k1 = 0.06                    -- update strength of 'qf' circuit
print(qf1.k1)                   -- display: 0.06 (lookup)
qf1.k1 = 0.07                   -- set strength of 'qf1' element
print(qf.k1, qf1.k1)            -- display: 0.06 0.07 (no lookup)
qf1.k1 = nil                    -- cancel strength of 'qf1' element
print(qf1.k1, qf1.l)            -- display: 0.06 2.1 (lookup)
print(#element:get_varkeys())   -- display: 33 (may vary)
```

The element `quadrupole` provided by the *element* module is the father of the objects created on its left. The *black arrows* show the user defined hierarchy of object created from and linked to the `quadrupole`. The main quadrupole `mq` is a user class representing the physical element, e.g. defining a length, and used to create two new classes, a focusing quadrupole `qf` and a defocusing quadrupole `qd` to model the circuits, e.g. hold the strength of elements connected in series, and finally the real individual elements `qf1`, `qd1`, `qf2` and `qd2` that will populate the sequence. A tracking command will request various attributes when crossing an element, like its length or its strength, leading to lookup of different depths in the hierarchy along the *red arrow*. A user may also write or overwrite an attribute at different level in the hierarchy by accessing directly to an element, as shown by the *purple arrows*, and mask an attribute of the parent with the new definitions in the children. The construction shown in this example follows the *separation of concern* principle and it is still highly reconfigurable despite that is does not contain any deferred expression or lambda function.

3 Attributes

New attributes can be added to objects using the dot operator `.` or the indexing operator `[]` as for tables. Attributes with non-*string* keys are considered as private. Attributes with *string* keys starting by two underscores are considered as private and read-only, and must be set during creation:

```
mq.comment = "Main Arc Quadrupole"
print(qf1.comment)      -- displays: Main Arc Quadrupole
qf.__k1 = 0.01          -- error
qf2 = qf { __k1=0.01 }  -- ok
```

The root `object` provides the following attributes:

name
: A *lambda* returning the *string* `__id`.

parent
: A *lambda* returning a *reference* to the parent *object*.

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

__id
A *string* holding the object's name set during its creation.

__par
A *reference* holding the object's parent set during its creation.

__flg
A *number* holding the object's flags.

__var
A *table* holding the object's variables, i.e. pairs of (*key*, *value*).

__env
A *table* holding the object's environment.

__index
A *reference* to the object's parent variables.

4 Methods

New methods can be added to objects but not classes, using the `:set_methods(set)` *method* with `set` being the *set* of methods to add as in the following example:

```
sequence :set_methods {
  name_of   = name_of,
  index_of  = index_of,
  range_of  = range_of,
  length_of = length_of,
  ...
}
```

where the keys are the names of the added methods and their values must be a *callable* accepting the object itself, i.e. `self`, as their first argument. Classes cannot set new methods.

The root `object` provides the following methods:

is_final
A *method* `()` returning a *boolean* telling if the object is final, i.e. cannot have instance.

is_class
A *method* `()` returning a *boolean* telling if the object is a *class*, i.e. had/has an instance.

is_readonly
A *method* `()` returning a *boolean* telling if the object is read-only, i.e. attributes cannot be changed.

is_instanceOf
A *method* `(cls)` returning a *boolean* telling if `self` is an instance of `cls`.

set_final
A *method* `([a])` returning `self` set as final if `a ~= false` or non-final.

set_readonly
A *method* `([a])` returning `self` set as read-only if `a ~= false` or read-write.

same
A *method* `([name])` returning an empty clone of `self` and named after the *string* `name` (default: `nil`).

copy
A *method* `([name])` returning a copy of `self` and named after the *string* `name` (default: `nil`). The private attributes are not copied, e.g. the final, class or read-only qualifiers are not copied.

get_varkeys
A *method* `([cls])` returning both, the *list* of the non-private attributes of `self` down to `cls` (default: `self`) included, and the *set* of their keys in the form of pairs (*key*, *key*).

get_variables
A *method* `(lst, [set], [noeval])` returning a *set* containing the pairs (*key*, *value*) of the attributes listed in `lst`. If `set` is provided, it will be used to store the pairs. If `noveval == true`, the functions are not evaluated. The full *list* of attributes can be retrieved from `get_varkeys`. Shortcut `getvar`.

set_variables
A *method* `(set, [override])` returning `self` with the attributes set to the pairs (*key*, *value*) contained in `set`. If `override ~= true`, the read-only attributes (with *key* starting by "`__`") cannot be updated.

copy_variables
A *method* `(set, [lst], [override])` returning `self` with the attributes listed in `lst` set to the pairs (*key*, *value*) contained in `set`. If `lst` is not provided, it is replaced by `self.__attr`. If `set` is an *object* and `lst.noeval` exists, it is used as the list of attributes to copy without function evaluation.[3] If `override ~= true`, the read-only attributes (with *key* starting by "`__`") cannot be updated. Shortcut `cpyvar`.

wrap_variables
A *method* `(set, [override])` returning `self` with the attributes wrapped by the pairs (*key*, *value*) contained in `set`, where the *value* must be a *callable* `(a)` that takes the attribute (as a callable) and returns the wrapped *value*. If `override ~= true`, the read-only attributes (with *key* starting by "`__`") cannot be updated.

The following example shows how to convert the length `l` of an RBEND from cord to arc,[4] keeping its strength `k0` to be computed on the fly:

```
local cord2arc in MAD.gmath
local rbend    in MAD.element
local printf   in MAD.utility
local rb = rbend 'rb' { angle=pi/10, l=2, k0=\s s.angle/s.l }
printf("l=%.5f, k0=%.5f\n", rb.l, rb.k0) -- l=2.00000, k0=0.15708
rb:wrap_variables { l=\l\s cord2arc(l(),s.angle) } -- RBARC
printf("l=%.5f, k0=%.5f\n", rb.l, rb.k0) -- l=2.00825, k0=0.15643
rb.angle = pi/20 -- update angle
printf("l=%.5f, k0=%.5f\n", rb.l, rb.k0) -- l=2.00206, k0=0.07846
```

The method converts non-*callable* attributes into callables automatically to simplify the user-side, i.e. `l()` can always be used as a *callable* whatever its original form was. At the end, `k0` and `l` are computed values and updating `angle` affects both as expected.

clear_variables
A *method* `()` returning `self` after setting all non-private attributes to `nil`.

clear_array

[3] This feature is used to setup a command from another command, e.g. `track` from `twiss`
[4] This approach is safer than the volatile option `RBARC` of MAD-X.

A *method* `()` returning `self` after setting the array slots to `nil`, i.e. clear the *list* part.

clear_all
A *method* `()` returning `self` after clearing the object except its private attributes.

set_methods
A *method* `(set, [override])` returning `self` with the methods set to the pairs (*key*, *value*) contained in `set`, where *key* must be a *string* (the method's name) and *value* must be a *callable* (the method itself). If `override ~= true`, the read-only methods (with *key* starting by `"__"`) cannot be updated. Classes cannot update their methods.

set_metamethods
A *method* `(set, [override])` returning `self` with the attributes set to the pairs (*key*, *value*) contained in `set`, where *key* must be a *string* (the metamethod's name) and *value* must be a *callable*(the metamethod itself). If `override == false`, the metamethods cannot be updated. Classes cannot update their metamethods.

insert
A *method* `([idx], a)` returning `self` after inserting `a` at the position `idx` (default: `#self+1`) and shifting up the items at positions `idx..`.

remove
A *method* `([idx])` returning the *value* removed at the position `idx` (default: `#self`) and shifting down the items at positions `idx..`.

move
A *method* `(idx1, idx2, idxto, [dst])` returning the destination object `dst` (default: `self`) after moving the items from `self` at positions `idx1..idx2` to `dst` at positions `idxto..`. The destination range can overlap with the source range.

sort
A *method* `([cmp])` returning `self` after sorting in-place its *list* part using the ordering *callable* (`cmp(ai, aj)`) (default: "<"), which must define a partial order over the items. The sorting algorithm is not stable.

bsearch
A *method* `(a, [cmp], [low], [high])` returning the lowest index `idx` in the range specified by `low..high` (default: `1..#self`) from the **ordered** *list* of `self` that compares `true` with item `a` using the *callable* (`cmp(a, self[idx])`) (default: "<=" for ascending, ">=" for descending), or `high+1`. In the presence of multiple equal items, "<=" (resp. ">=") will return the index of the first equal item while "<" (resp. ">") the index next to the last equal item for ascending (resp. descending) order.[5]

lsearch
A *method* `(a, [cmp], [low], [high])` returning the lowest index `idx` in the range specified by `low..high` (default: `1..#self`) from the *list* of `self` that compares `true` with item `a` using the *callable* (`cmp(a, self[idx])`) (default: "=="), or `high+1`. In the presence of multiple equal items in an ordered *list*, "<=" (resp. ">=") will return the index of the first equal item while "<" (resp. ">") the index next to the last equal item for ascending (resp. descending) order.Page 34, 5

get_flags
A *method* `()` returning the flags of `self`. The flags are not inherited nor copied.

set_flags
A *method* `(flgs)` returning `self` after setting the flags determined by `flgs`.

[5] `bsearch` and `lsearch` stand for binary (ordered) search and linear (unordered) search respectively.

clear_flags
A *method* `(flgs)` returning `self` after clearing the flags determined by `flgs`.

test_flags
A *method* `(flgs)` returning a *boolean* telling if all the flags determined by `flgs` are set.

open_env
A *method* `([ctx])` returning `self` after opening an environment, i.e. a global scope, using `self` as the context for `ctx` (default: 1). The argument `ctx` must be either a *function* or a *number* defining a call level ≥ 1.

close_env
A *method* `()` returning `self` after closing the environment linked to it. Closing an environment twice is safe.

load_env
A *method* `(loader)` returning `self` after calling the `loader`, i.e. a compiled chunk, using `self` as its environment. If the loader is a *string*, it is interpreted as the filename of a script to load, see functions `load` and `loadfile` in Lua 5.2 §6.1 for details.

dump_env
A *method* `()` returning `self` after dumping its content on the terminal in the rought form of pairs (*key*, *value*), including content of table and object *value*, useful for debugging environments.

is_open_env
A *method* `()` returning a *boolean* telling if `self` is an open environment.

raw_len
A *method* `()` returning the *number* of items in the *list* part of the object. This method should not be confused with the native *function* `rawlen`.

raw_get
A *method* `(key)` returning the *value* of the attribute `key` without *lambda* evaluation nor inheritance lookup. This method should not be confused with the native *function* `rawget`.

raw_set
A *method* `(key, val)` setting the attribute `key` to the *value* `val`, bypassing all guards of the object model. This method should not be confused with the native *function* `rawset`. **Warning**: use this dangerous method at your own risk!

var_get
A *method* `(key)` returning the *value* of the attribute `key` without *lambda* evaluation.

var_val
A *method* `(key, val)` returning the *value* `val` of the attribute `key` with *lambda* evaluation. This method is the complementary of `var_get`, i.e. `__index` $\equiv$ `var_val` $\circ$ `var_get`.

dumpobj
A *method* `([fname], [cls], [patt], [noeval])` return `self` after dumping its non-private attributes in file `fname` (default: `stdout`) in a hierarchical form down to `cls`. If the *string* `patt` is provided, it filters the names of the attributes to dump. If `fname == '-'`, the dump is returned as a *string* in place of `self`. The *logical* `noeval` prevents the evaluatation the deferred expressions and reports the functions addresses instead. In the output, `self` and its parents are displayed indented according to their inheritance level, and preceeded by a + sign. The attributes overridden through the inheritance are tagged with n * signs, where n corresponds to the number of overrides since the first definition.

5 Metamethods

New metamethods can be added to objects but not classes, using the `:set_metamethods(set)` *method* with `set` being the *set* of metamethods to add as in the following example:

```
sequence :set_metamethods {
  __len      = len_mm,
  __index    = index_mm,
  __newindex = newindex_mm,
  ...
}
```

where the keys are the names of the added metamethods and their values must be a *callable* accepting the object itself, i.e. `self`, as their first argument. Classes cannot set new metamethods.

The root `object` provides the following metamethods:

__init
: A *metamethod* `()` called to finalize `self` before returning from the constructor.

__same
: A *metamethod* `()` similar to the *method* `same`.

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

__len
: A *metamethod* `()` called by the length operator `#` to return the size of the *list* part of `self`.

__call
: A *metamethod* `([name], tbl)` called by the call operator `()` to return an instance of `self` created from `name` and `tbl`, i.e. using `self` as a constructor.

__index
: A *metamethod* `(key)` called by the indexing operator `[key]` to return the *value* of an attribute determined by *key* after having performed *lambda* evaluation and inheritance lookup.

__newindex
: A *metamethod* `(key, val)` called by the assignment operator `[key]=val` to create new attributes for the pairs (*key*, *value*).

__pairs
: A *metamethod* `()` called by the `pairs` *function* to return an iterator over the non-private attributes of `self`.

__ipairs
: A *metamethod* `()` called by the `ipairs` *function* to return an iterator over the *list* part of `self`.

__tostring
: A *metamethod* `()` called by the `tostring` *function* to return a *string* describing succinctly `self`.

The following attributes are stored with metamethods in the metatable, but have different purposes:

__obj
: A unique private *reference* that characterizes objects.

__metatable

A *reference* to the metatable itself protecting against modifications.

6 Flags

The object model uses *flags* to qualify objects, like *class*-object, *final*-object and *readonly*-object. The difference with *boolean* attributes is that flags are *not* inherited nor copied. The flags of objects are managed by the methods `:get_flags`, `:set_flags`, `:clear_flags` and `:test_flags`. Methods like `:is_class`, `:is_final` and `:is_readonly` are roughly equivalent to call the method `:test_flags` with the corresponding (private) flag as argument. Note that functions from the `typeid` module that check for types or kinds, like `is_object` or `is_beam`, never rely on flags because types and kinds are not qualifers.

From the technical point of view, flags are encoded into a 32-bit integer and the object model uses the protected bits 29-31, hence bits 0-28 are free of use. Object flags can be used and extended by other modules introducing their own flags, like the `element` module that relies on bits 0-4 and used by many commands. In practice, the bit index does not need to be known and should not be used directly but through its name to abstract its value.

7 Environments

The object model allows to transform an object into an environment; in other words, a global workspace for a given context, i.e. scope. Objects-as-environments are managed by the methods `open_env`, `close_env`, `load_env`, `dump_env` and `is_open_env`. Things defined in this workspace will be stored in the object, and accessible from outside using the standard ways to access object attributes:

```
local object in MAD
local one = 1
local obj = object { a:=one } -- obj with 'a' defined
-- local a = 1                -- see explication below

obj:open_env()                -- open environment
b = 2                         -- obj.b defined
c =\ -> a..":"..b             -- obj.c defined
obj:close_env()               -- close environment

print(obj.a, obj.b, obj.c)    -- display: 1   2   1:2
one = 3
print(obj.a, obj.b, obj.c)    -- display: 3   2   3:2
obj.a = 4
print(obj.a, obj.b, obj.c)    -- display: 4   2   4:2
```

Uncommenting the line `local a = 1` would change the last displayed column to `1:2` for the three prints because the *lambda* defined for `obj.c` would capture the local `a` as it would exist in its scope. As seen hereabove, once the environment is closed, the object still holds the variables as attributes.

The MADX environment is an object that relies on this powerful feature to load MAD-X lattices, their settings and their "business logic", and provides functions, constants and elements to mimic the behavior of the global workspace of MAD-X to some extend:

```
MADX:open_env()
mq_k1 = 0.01                        -- mq.k1 is not a valid identifier!
MQ = QUADRUPOLE {l=1, k1:=MQ_K1} -- MADX environment is case insensitive
MADX:close_env()                    -- but not the attributes of objects!
local mq in MADX
print(mq.k1)                        -- display: 0.01
MADX.MQ_K1 = 0.02
print(mq.k1)                        -- display: 0.02
```

Note that MAD-X workspace is case insensitive and everything is "global" (no scope, namespaces), hence the `quadrupole` element has to be directly available inside the MADX environment. Moreover, the MADX object adds the method `load` to extend `load_env` and ease the conversion of MAD-X lattices. For more details see *MADX*.

Chapter 4. Beams

The `beam` object is the *root object* of beams that store information relative to particles and particle beams. It also provides a simple interface to the particles and nuclei database.

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

1 Attributes

The `beam` *object* provides the following attributes:

particle
: A *string* specifying the name of the particle. (default: `"positron"`).

mass
: A *number* specifying the energy-mass of the particle [GeV]. (default: `emass`).

charge
: A *number* specifying the charge of the particle in [q] unit of `qelect`.[1] (default: `1`).

spin
: A *number* specifying the spin of the particle. (default: `0`).

emrad
: A *lambda* returning the electromagnetic radius of the particle [m],
$\mathrm{emrad} = \mathrm{krad_G eV} \times \mathrm{charge}^2/\mathrm{mass}$ where $\mathrm{krad_G eV} = 10^{-9}(4\pi\varepsilon_0)^{-1}q$.

aphot
: A *lambda* returning the average number of photon emitted per bending unit,
$\mathrm{aphot} = \mathrm{kpht_G eV} \times \mathrm{charge}^2 \times \mathrm{betgam}$ where $\mathrm{kpht_G eV} = \frac{5}{2\sqrt{3}}\,\mathrm{krad_G eV}\,(\hbar c)^{-1}$.

energy
: A *number* specifying the particle energy [GeV]. (default: `1`).

pc
: A *lambda* returning the particle momentum times the speed of light [GeV],
$\mathrm{pc} = (\mathrm{energy}^2 - \mathrm{mass}^2)^{\frac{1}{2}}$.

beta
: A *lambda* returning the particle relativistic $\beta = \frac{v}{c}$,
$\mathrm{beta} = (1 - (\mathrm{mass}/\mathrm{energy})^2)^{\frac{1}{2}}$.

gamma
: A *lambda* returning the particle Lorentz factor $\gamma = (1 - \beta^2)^{-\frac{1}{2}}$,
$\mathrm{gamma} = \mathrm{energy}/\mathrm{mass}$.

betgam
: A *lambda* returning the product $\beta\gamma$,
$\mathrm{betgam} = (\mathrm{gamma}^2 - 1)^{\frac{1}{2}}$.

[1] The `qelect` value is defined in the *Constants* module.

pc2
A *lambda* returning pc^2, avoiding the square root.

beta2
A *lambda* returning beta^2, avoiding the square root.

betgam2
A *lambda* returning betgam^2, avoiding the square root.

brho
A *lambda* returning the magnetic rigidity [T.m],
`brho = GeV_c * pc/|charge|` where `GeV_c` = $10^9/c$

ex
A *number* specifying the horizontal emittance ϵ_x [m]. (default: `1`).

ey
A *number* specifying the vertical emittance ϵ_y [m]. (default: `1`).

et
A *number* specifying the longitudinal emittance ϵ_t [m]. (default: `1e-3`).

exn
A *lambda* returning the normalized horizontal emittance [m],
`exn = ex * betgam`.

eyn
A *lambda* returning the normalized vertical emittance [m],
`eyn = ey * betgam`.

etn
A *lambda* returning the normalized longitudinal emittance [m],
`etn = et * betgam`.

nbunch
A *number* specifying the number of particle bunches in the machine. (default: `0`).

npart
A *number* specifying the number of particles per bunch. (default: `0`).

sigt
A *number* specifying the bunch length in $c\sigma_t$. (default: `1`).

sige
A *number* specifying the relative energy spread in σ_E/E [GeV]. (default: `1e-3`).

The `beam` *object* also implements a special protect-and-update mechanism for its attributes to ensure consistency and precedence between the physical quantities stored internally:

- The following attributes are *read-only*, i.e. writing to them triggers an error:
  `mass, charge, spin, emrad, aphot`
- The following attributes are *read-write*, i.e. hold values, with their accepted numerical ranges:
  `particle, energy` $>$ `mass`, `ex` > 0, `ey` > 0, `et` > 0, `nbunch` > 0, `npart` > 0, `sigt` > 0, `sige` > 0.
- The following attributes are *read-update*, i.e. setting these attributes update the `energy`, with their accepted numerical ranges:
  `pc` > 0, $0.9 >$ `beta` > 0, `gamma` > 1, `betgam` > 0.1, `brho` > 0, `pc2`, `beta2`, `betgam2`.

– The following attributes are *read-update*, i.e. setting these attributes update the emittances `ex`, `ey`, and `et` repectively, with their accepted numerical ranges:
`exn` > 0, `eyn` > 0, `etn` > 0.

2 Methods

The `beam` object provides the following methods:

new_particle
A *method* `(particle, mass, charge, [spin])` creating new particles or nuclei and store them in the particles database. The arguments specify in order the new `particle`'s name, energy-`mass` [GeV], `charge` [q], and `spin` (default: `0`). These arguments can also be grouped into a *table* with same attribute names as the argument names and passed as the solely argument.

set_variables
A *method* `(set)` returning `self` with the attributes set to the pairs (*key*, *value*) contained in `set`. This method overrides the original one to implement the special protect-and-update mechanism, but the order of the updates is undefined. It also creates new particle on-the-fly if the `mass` and the `charge` are defined, and then select it. Shortcut `setvar`.

showdb
A *method* `([file])` displaying the content of the particles database to `file` (default: `io.stdout`).

3 Metamethods

The `beam` object provides the following metamethods:

__init
A *metamethod* `()` returning `self` after having processed the attributes with the special protect-and-update mechanism, where the order of the updates is undefined. It also creates new particle on-the-fly if the `mass` and the `charge` are defined, and then select it.

__newindex
A *metamethod* `(key, val)` called by the assignment operator `[key]=val` to create new attributes for the pairs (*key*, *value*) or to update the underlying physical quantity of the `beam` objects.

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

__beam
A unique private *reference* that characterizes beams.

4 Particles database

The `beam` *object* manages the particles database, which is shared by all `beam` instances. The default set of supported particles is:

electron, positron, proton, antiproton, neutron, antineutron, ion, muon, antimuon, deuteron, antideuteron, negmuon (=muon), posmuon (=antimuon).

New particles can be added to the database, either explicitly using the `new_particle` method, or by creating or updating a beam *object* and specifying all the attributes of a particle, i.e. `particle`'s name, `charge`, `mass`, and (optional) `spin`:

```
local beam in MAD
local nmass, pmass, mumass in MAD.constant

-- create a new particle
beam:new_particle{ particle='mymuon', mass=mumass, charge=-1, spin=1/2 }

-- create a new beam and a new nucleus
local pbbeam = beam { particle='pb208', mass=82*pmass+126*nmass, charge=82 }
```

The particles database can be displayed with the `showdb` method at any time from any beam:

```
beam:showdb()  -- check that both, mymuon and pb208 are in the database.
```

5 Particle charges

The physics of MAD-NG is aware of particle charges. To enable the compatibility with codes like MAD-X that ignores the particle charges, the global option `nocharge` can be used to control the behavior of created beams as shown by the following example:

```
local beam, option in MAD
local beam1 = beam { particle="electron" } -- beam with negative charge
print(beam1.charge, option.nocharge)       -- display: -1  false

option.nocharge = true                     -- disable particle charges
local beam2 = beam { particle="electron" } -- beam with negative charge
print(beam2.charge, option.nocharge)       -- display:  1  true

-- beam1 was created before nocharge activation...
print(beam1.charge, option.nocharge)       -- display: -1  true
```

This approach ensures consistency of beams behavior during their entire lifetime.[2]

[2] The option `rbarc` in MAD-X is too volatile and does not ensure such consistency...

6 Examples

The following code snippet creates the LHC lead beams made of bare nuclei $^{208}\mathrm{Pb}^{82+}$

```
local beam in MAD
local lhcb1, lhcb2 in MADX
local nmass, pmass, amass in MAD.constant
local pbmass = 82*pmass+126*nmass

-- attach a new beam with a new particle to lhcb1 and lhcb2.
lhc1.beam = beam 'Pb208' { particle='pb208', mass=pbmass, charge=82 }
lhc2.beam = lhc1.beam -- let sequences share the same beam...

-- print Pb208 nuclei energy-mass in GeV and unified atomic mass.
print(lhcb1.beam.mass, lhcb1.beam.mass/amass)
```

Chapter 5. Beta0 Blocks

The `beta0` object is the *root object* of beta0 blocks that store information relative to the phase space at given positions, e.g. initial conditions, Poincaré section.

The `beta0` module extends the *typeid* module with the `is_beta0` *function*, which returns `true` if its argument is a `beta0` object, `false` otherwise.

1 Attributes

The `beta0` *object* provides the following attributes:

particle
: A *string* specifying the name of the particle. (default: `"positron"`).

2 Methods

The `beta0` object provides the following methods:

showdb
: A *method* `([file])` displaying the content of the particles database to `file` (default: `io.stdout`).

3 Metamethods

The `beta0` object provides the following metamethods:

__init
: A *metamethod* `()` returning `self` after having processed the attributes with the special protect-and-update mechanism, where the order of the updates is undefined. It also creates new particle on-the-fly if the `mass` and the `charge` are defined, and then select it.

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

__beta0
: A unique private *reference* that characterizes beta0 blocks.

4 Examples

Chapter 6. Elements

The `element` object is the *root object* of all elements used to model particle accelerators, including sequences and drifts. It provides most default values inherited by all elements.

The `element` module extends the *typeid* module with the `is_element` *function*, which returns `true` if its argument is an `element` object, `false` otherwise.

1 Taxonomy

The classes defined by the `element` module are organized according to the kinds and the roles of their instances. The classes defining the kinds are:

thin
: The *thin* elements have zero-length and their physics does not depend on it, i.e. the attribute `l` is discarded or forced to zero in the physics.

thick
: The *thick* elements have a length and their physics depends on it. Elements like `sbend`, `rbend`, `quadrupole`, `solenoid`, and `elseparator` trigger a runtime error if they have zero-length. Other thick elements will accept to have zero-length for compatibility with MAD-X[1] , but their physics will have to be adjusted.[2]

drift
: The *drift* elements have a length with a `drift`-like physics if $l \geq$ `minlen`[3] otherwise they are discarded or ignored. Any space between elements with a length $l \geq$ `minlen` are represented by an `implicit` drift created on need by the s-iterator of sequences and discarded afterward.

patch
: The *patch* elements have zero-length and the purpose of their physics is to change the reference frame.

extrn
: The *extern* elements are never part of sequences. If they are present in a sequence definition, they are expanded and replaced by their content, i.e. stay external to the lattice.

specl
: The *special* elements have special roles like *marking* places (i.e. `maker`) or *branching* sequences (i.e. `slink`).

These classes are not supposed to be used directly, except for extending the hierarchy defined by the `element` module and schematically reproduced hereafter to help users understanding:

```
thin_element = element  'thin_element' { is_thin    = true }
thick_element = element 'thick_element' { is_thick   = true }
drift_element = element 'drift_element' { is_drift   = true }
patch_element = element 'patch_element' { is_patch   = true }
extrn_element = element 'extrn_element' { is_extern  = true }
```

(continues on next page)

[1] In MAD-X, zero-length `sextupole` and `octupole` are valid but may have surprising effects…
[2] E.g. zero-length `sextupole` must define their strength with `knl[3]` instead of `k2` to have the expected effect.
[3] By default `minlen` = 10^{-12} m.

(continued from previous page)

```
specl_element = element 'specl_element' { is_special = true }

sequence    = extrn_element 'sequence'    { }
assembly    = extrn_element 'assembly'    { }
bline       = extrn_element 'bline'       { }

marker      = specl_element 'marker'      { }
slink       = specl_element 'slink'       { }

drift       = drift_element 'drift'       { }
collimator  = drift_element 'collimator'  { }
instrument  = drift_element 'instrument'  { }
placeholder = drift_element 'placeholder' { }

sbend       = thick_element 'sbend'       { }
rbend       = thick_element 'rbend'       { }
quadrupole  = thick_element 'quadrupole'  { }
sextupole   = thick_element 'sextupole'   { }
octupole    = thick_element 'octupole'    { }
decapole    = thick_element 'decapole'    { }
dodecapole  = thick_element 'dodecapole'  { }
solenoid    = thick_element 'solenoid'    { }
tkicker     = thick_element 'tkicker'     { }
wiggler     = thick_element 'wiggler'     { }
elseparator = thick_element 'elseparator' { }
rfcavity    = thick_element 'rfcavity'    { }
genmap      = thick_element 'genmap'      { }

beambeam    = thin_element  'beambeam'    { }
multipole   = thin_element  'multipole'   { }

xrotation   = patch_element 'xrotation'   { }
yrotation   = patch_element 'yrotation'   { }
srotation   = patch_element 'srotation'   { }
translate   = patch_element 'translate'   { }
changeref   = patch_element 'changeref'   { }
changedir   = patch_element 'changedir'   { }
changenrj   = patch_element 'changenrj'   { }

-- specializations
rfmultipole = rfcavity      'rfmultipole' { }
crabcavity  = rfmultipole   'crabcavity'  { }

monitor     = instrument     'monitor'    { }
hmonitor    = monitor       'hmonitor'    { }
```

(continues on next page)

(continued from previous page)

```
vmonitor    = monitor       'vmonitor'    { }

kicker      = tkicker        'kicker'     { }
hkicker     =  kicker       'hkicker'     { }
vkicker     =  kicker       'vkicker'     { }
```

All the classes above, including `element`, define the attributes `kind = name` and `is_name = true` where `name` correspond to the class name. These attributes help to identify the kind and the role of an element as shown in the following code excerpt:

```
local drift, hmonitor, sequence in MAD.element
local dft = drift    {}
local bpm = hmonitor {}
local seq = sequence {}
print(dft.kind)               -- display: drift
print(dft.is_drift)           -- display: true
print(dft.is_drift_element)   -- display: true
print(bpm.kind)               -- display: hmonitor
print(bpm.is_hmonitor)        -- display: true
print(bpm.is_monitor)         -- display: true
print(bpm.is_instrument)      -- display: true
print(bpm.is_drift_element)   -- display: true
print(bpm.is_element)         -- display: true
print(bpm.is_drift)           -- display: true
print(bpm.is_thick_element)   -- display: nil (not defined = false)
print(seq.kind)               -- display: sequence
print(seq.is_element)         -- display: true
print(seq.is_extrn_element)   -- display: true
print(seq.is_thick_element)   -- display: nil (not defined = false)
```

2 Attributes

The `element` *object* provides the following attributes:

l

A *number* specifying the physical length of the element on the design orbit [m]. (default: `0`).

lrad

A *number* specifying the field length of the element on the design orbit considered by the radiation [m]. (default: `lrad = \s -> s.l`).

angle

A *number* specifying the bending angle α of the element [rad]. A positive angle represents a bend to the right, i.e. a $-y$-rotation towards negative x values. (default: `0`).

tilt

A *number* specifying the physical tilt of the element [rad]. All the physical quantities defined by the element are in the tilted frame, except `misalign` that comes first when tracking through an element,

see the *track* command for details. (default: `0`).

model
A *string* specifying the integration model `"DKD"` or `"TKT"` to use when tracking through the element and overriding the command attribute, see the *track* command for details. (default: `cmd.model`).

method
A *number* specifying the integration order 2, 4, 6, or 8 to use when tracking through the element and overriding the command attribute, see the *track* command for details. (default: `cmd.method`).

nslice
A *number* specifying the number of slices or a *list* of increasing relative positions or a *callable* `(elm, mflw, lw)` returning one of the two previous kind of positions specification to use when tracking through the element and overriding the command attribute, see the *survey* or the *track* commands for details. (default: `cmd.nslice`).

refpos
A *string* holding one of `"entry"`, `"centre"` or `"exit"`, or a *number* specifying a position in [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 when inserted in a sequence, see *element positions* for details. (default: `nil` $\equiv$ `seq.refer`).

aperture
A *mappable* specifying aperture attributes, see *Aperture* for details. (default: `{kind='circle', 1}`).

apertype
A *string* specifying the aperture type, see *Aperture* for details. (default: `\s -> s.aperture.kind or 'circle'`).[4]

misalign
A *mappable* specifying misalignment attributes, see *Misalignment* for details. (default: `nil`)

The `thick_element` *object* adds the following multipolar and fringe fields attributes:

knl, ksl
A *list* specifying respectively the **multipolar** and skew integrated strengths of the element [m^{-i+1}]. (default: `{}`).

dknl, dksl
A *list* specifying respectively the multipolar and skew integrated strengths errors of the element [m^{-i+1}]. (default: `{}`).

e1, e2
A *number* specifying respectively the horizontal angle of the pole faces at entry and exit of the element [rad]. A positive angle goes toward inside the element, see Fig. 6.1 and Fig. 6.2. (default: `0`).

h1, h2
A *number* specifying respectively the horizontal curvature of the pole faces at entry and exit of the element [m^{-1}]. A positive curvature goes toward inside the element. (default: `0`).

hgap
A *number* specifying half of the vertical gap at the center of the pole faces of the element [m]. (default: `0`).

fint
A *number* specifying the fringe field integral at entrance of the element. (default: `0`).

[4] This attribute was introduced to ease the translation of MAD-X sequences and may disappear in some future.

fintx
A *number* specifying the fringe field integral at exit of the element. (default: `fint`).

fringe
A *number* specifying the bitmask to activate fringe fields of the element, see *Flags* for details. (default: `0`).

fringemax
A *number* specifying the maximum order for multipolar fringe fields of the element. (default: `2`).

kill_ent_fringe
A *logical* specifying to kill the entry fringe fields of the element. (default: `false`).

kill_exi_fringe
A *logical* specifying to kill the entry fringe fields of the element. (default: `false`).

f1, f2
A *number* specifying quadrupolar fringe field first and second parameter of SAD. (default: `0`).

3 Methods

The `element` object provides the following methods:

select
A *method* `([flg])` to select the element for the flags `flg` (default: `selected`).

deselect
A *method* `([flg])` to deselect the element for the flags `flg` (default: `selected`).

is_selected
A *method* `([flg])` to test the element for the flags `flg` (default: `selected`).

is_disabled
A *method* `()` to test if the element is *disabled*, which is equivalent to call the method `is_selected(disabled)`.

is_observed
A *method* `()` to test if the element is *observed*, which is equivalent to call the method `is_selected(observed)`.

is_implicit
A *method* `()` to test if the element is *implicit*, which is equivalent to call the method `is_selected(implicit)`.

The `drift_element` and `thick_element` objects provide the following extra methods, see *sub-elements* for details about the `sat` attribute:

index_sat
A *method* `(sat, [cmp])` returning the lowest index `idx` (starting from 1) of the first sub-element with a relative position from the element entry that compares `true` with the *number* `sat` using the optional *callable* `cmp(sat, self[idx].sat)` (default: `"=="`), or `#self+1`. In the presence of multiple equal positions, `"<="` (resp. `">="`) will return the lowest index of the position while `"<"` (resp. `">"`) the lowest index next to the position for ascending (resp. descending) order.

insert_sat
A *method* `(elm, [cmp])` returning the element after inserting the sub-element `elm` at the index de-

termined by `:index_sat(elm.sat, [cmp])` using the optional *callable* `cmp` (default: `"<"`).

replace_sat
A *method* `(elm)` returning the replaced sub-element found at the index determined by `:index_sat(elm.sat)` by the new sub-element `elm`, or `nil`.

remove_sat
A *method* `(sat)` returning the removed sub-element found at the index determined by `:index_sat(sat)`, or `nil`.

4 Metamethods

The `element` object provides the following metamethods:

__len
A *metamethod* `()` overloading the length operator # to return the number of subelements in the *list* part of the element.

__add
A *metamethod* `(obj)` overloading the binary operator + to build a `bline` object from the juxtaposition of two elements.

__mul
A *metamethod* `(n)` overloading the binary operator * to build a `bline` object from the repetition of an element `n` times, i.e. one of the two operands must be a *number*.

__unm
A *metamethod* `(n)` overloading the unary operator - to build a `bline` object from the turning of an element, i.e. reflect the element.

__tostring
A *metamethod* `()` returning a *string* built from the element information, e.g. `print(monitor 'bpm' {})` display the *string* `":monitor: 'bpm' memory-address"`.

The operators overloading of elements allows to unify sequence and beamline definitions in a consistent and simple way, noting that `sequence` and `bline` are (external) elements too.

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

__elem
A unique private *reference* that characterizes elements.

5 Elements

Some elements define new attributes or override the default values provided by the *root object* `element`. The following subsections describe the elements supported by MAD-NG.

5.1 SBend

The `sbend` element is a sector bending magnet with a curved reference system as shown in Fig. 6.1, and defines or overrides the following attributes:

k0

A `number` specifying the dipolar strength of the element [m^{-1}]. (default: `k0 = \s -> s.angle/s.l`).[5][6]

k0s

A *number* specifying the dipolar skew strength of the element [m^{-1}]. (default: `0`).

k1, k1s

A *number* specifying respectively the quadrupolar and skew strengths of the element [m^{-2}]. (default: `0`).

k2, k2s

A *number* specifying respectively the sextupolar and skew strengths of the element [m^{-3}]. (default: `0`).

fringe

Set to flag `fringe.bend` to activate the fringe fields by default, see *Flags* for details.

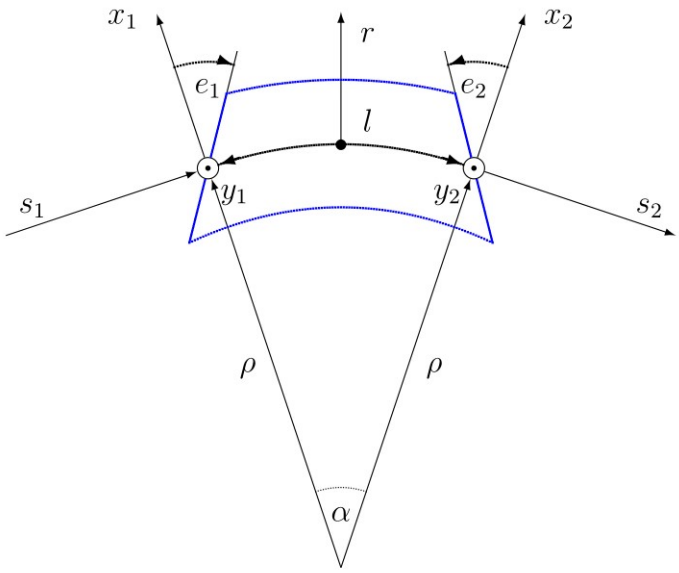

Figure6.1: Reference system for a sector bending magnet.

[5] By default bending magnets are ideal bends, that is `angle = k0*l`

[6] For compatibility with MAD-X.

5.2 RBend

The `rbend` element is a rectangular bending magnet with a straight reference system as shown in Fig. 6.2, and defines or overrides the following attributes:

k0

A `number` specifying the dipolar strength of the element [m^{-1}]. (default: `k0 = \s -> s.angle/s.l`).[Page 51, 5][Page 51, 6]

k0s

A *number* specifying the dipolar skew strength of the element [m^{-1}]. (default: `0`).

k1, k1s

A *number* specifying respectively the quadrupolar and skew strengths of the element [m^{-2}]. (default: `0`).

k2, k2s

A *number* specifying respectively the sextupolar and skew strengths of the element [m^{-3}]. (default: `0`).

fringe

Set to flag `fringe.bend` to activate the fringe fields by default, see *Flags* for details.

true_rbend

A *logical* specifying if this `rbend` element behaves like (`false`) a `sbend` element with parallel pole faces, i.e. $e_1 = e_2 = \alpha/2$ in Fig. 6.1 , or like (`true`) a rectangular bending magnet with a straight reference system as shown in Fig. 6.2. (default: `false`).[Page 51, 6]

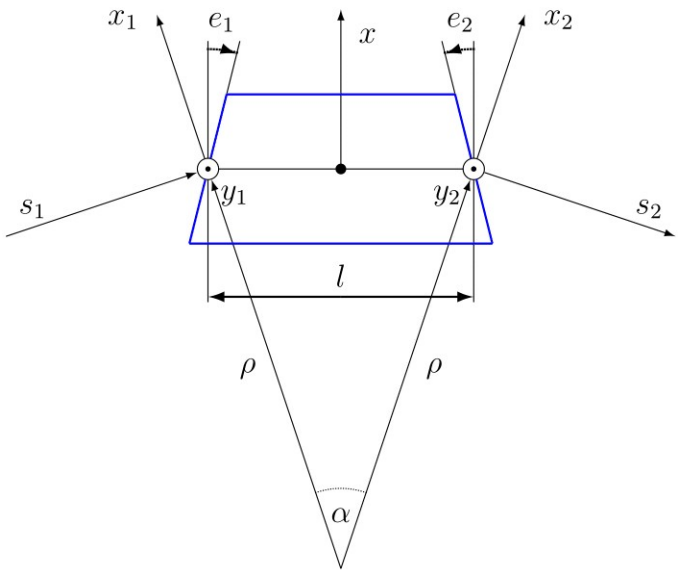

Figure6.2: Reference system for a rectangular bending magnet.

5.3 Quadrupole

The `quadrupole` element is a straight focusing element and defines the following attributes:

k0, k0s
A *number* specifying respectively the dipolar and skew strengths of the element [m^{-1}]. (default: `0`).

k1, k1s
A *number* specifying respectively the quadrupolar and skew strengths of the element [m^{-2}]. (default: `0`).

k2, k2s
A *number* specifying respectively the sextupolar and skew strengths of the element [m^{-3}]. (default: `0`).

5.4 Sextupole

The `sextupole` element is a straight element and defines the following attributes:

k2, k2s
A *number* specifying respectively the sextupolar and skew strengths of the element [m^{-3}]. (default: `0`).

5.5 Octupole

The `octupole` element is a straight element and defines the following attributes:

k3, k3s
A *number* specifying respectively the octupolar and skew strengths of the element [m^{-4}]. (default: `0`).

5.6 Decapole

The `decapole` element is a straight element and defines the following attributes:

k4, k4s
A *number* specifying respectively the decapolar and skew strength of the element [m^{-5}]. (default: `0`).

5.7 Dodecapole

The `dodecapole` element is a straight element and defines the following attributes:

k5, k5s
A *number* specifying respectively the dodecapolar and skew strength of the element [m^{-6}]. (default: `0`).

5.8 Solenoid

The `solenoid` element defines the following attributes:

ks, ksi
A *number* specifying respectively the strength [rad/m] and the integrated strength [rad] of the element. A positive value points toward positive s. (default: `0`).

5.9 Multipole

The `multipole` element is a thin element and defines the following attributes:

knl, ksl
A *list* specifying respectively the multipolar and skew integrated strengths of the element [m^{-i+1}]. (default: `{}`).

dknl, dksl
A *list* specifying respectively the multipolar and skew integrated strengths errors of the element [m^{-i+1}]. (default: `{}`).

5.10 TKicker

The `tkicker` element is the *root object* of kickers and defines or overrides the following attributes:

hkick
A *number* specifying the horizontal strength of the element [m^{-1}]. By convention, a kicker with a positive horizontal strength kicks in the direction of the reference orbit, e.g. `hkick` $\equiv$ `- knl[1]`. (default: `0`).

vkick
A *number* specifying the vertical strength of the element [m^{-1}]. By convention, a kicker with a positive vertical strength kicks toward the reference orbit, e.g. `vkick` $\equiv$ `ksl[1]`. (default: `0`).

method
Set to `2` if `ptcmodel` is not set to enforce pure momentum kick and avoid dipolar strength integration that would introduce dispersion.

5.11 Kicker, HKicker, VKicker

The `kicker` element inheriting from the `tkicker` element, is the *root object* of kickers involved in the orbit correction and defines the following attributes:

chkick, cvkick
A *number* specifying respectively the horizontal and vertical correction strength of the element set by the *correct* command [m^{-1}]. (default:).

The `hkicker` (horizontal kicker) and `vkicker` (vertical kicker) elements define the following attribute:

kick
A *number* specifying the strength of the element in its main direction [m^{-1}]. (default:).

5.12 Monitor, HMonitor, VMonitor

The `monitor` element is the root object of monitors involved in the orbit correction and defines the following attributes:

mredx, mredy
A *number* specifying respectively the readout x, y-offset error of the element [m]. The offset is added to the beam position during orbit correction (after scaling). (default: `0`).

mresx, mresy
A *number* specifying respectively the readout x, y-scaling error of the element. The scale factor multiplies the beam position by `1+mres` (before offset) during orbit correction.[7] (default: `0`).

The `hmonitor` (horizontal monitor) and `vmonitor` (vertical monitor) elements are specialisations inheriting from the `monitor` element.

5.13 RFCavity

The `rfcavity` element defines the following attributes:

volt
A *number* specifying the peak RF voltage of the element [MV]. (default: `0`).

freq
A *number* specifying a non-zero RF frequency of the element [MHz]. (default: `0`).

lag
A *number* specifying the RF phase lag of the element in unit of 2π. (default: `0`).

harmon
A *number* specifying the harmonic number of the element if `freq` is zero. (default: `0`).

n_bessel
A *number* specifying the transverse focussing effects order of the element. (default: `0`).

totalpath
A *logical* specifying if the totalpath must be used in the element. (default: `true`).

5.14 RFMultipole

The `rfmultipole` element defines the following attributes:

pnl, psl
A *list* specifying respectively the multipolar and skew phases of the element [rad]. (default: `{}`).

dpnl, dpsl
A *list* specifying respectively the multipolar and skew phases errors of the element [rad]. (default: `{}`).

[7] This definition comes from MAD-X default zeroed values such that undefined attribute gives a scale of `1`.

5.15 ElSeparator

The `elseparator` element defines the following attributes:

ex, ey
A *number* specifying respectively the electric field x, y-strength of the element [MV/m]. (default: `0`).

exl, eyl
A *number* specifying respectively the integrated electric field x, y-strength of the element [MV]. (default: `0`).

5.16 Wiggler

The `wiggler` element defines the following attributes: NYI, TBD

5.17 BeamBeam

The `beambeam` element defines the following attributes: NYI, TBD

5.18 GenMap

The `genmap` element defines the following attributes:[8]

damap
A `damap` used for thick integration.

update
A *callable* `(elm, mflw, lw)` invoked before each step of thick integration to update the `damap`. (default: `nil`)

nslice
A *number* specifying the number of slices or a *list* of increasing relative positions or a *callable* `(elm, mflw, lw)` returning one of the two previous kind of positions specification to use when tracking through the element and overriding the command attribute, see the *survey* or the *track* commands for details. (default: `1`).

5.19 SLink

The `slink` element defines the following attributes:[9]

sequence
A *sequence* to switch to right after exiting the element. (default: `nil`)

range
A *range* specifying the span over the sequence to switch to, as expected by the sequence method `:siter`. (default: `nil`).

[8] This element is a generalization of the `matrix` element of MAD-X, to use with care!
[9] This element allows to switch between sequences during tracking, kind of `if-then-else` for tracking.

nturn
A *number* specifying the number of turn to track the sequence to switch to, as expected by the sequence method `:siter`. (default: `nil`).

dir
A *number* specifying the s-direction of the tracking of the sequence to switch to, as expected by the sequence method `:siter`. (default: `nil`).

update
A *callable* `(elm, mflw)` invoked before retrieving the other attributes when entering the element. (default: `nil`)

5.20 Translate

The `translate` element is a patch element and defines the following attributes:

dx, dy, ds
A *number* specifying respectively x, y, s-translation of the reference frame [m]. (default: `0`)

5.21 XRotation, YRotation, SRotation

The `xrotation` (rotation around x-axis), `yrotation` (rotation around y-axis) and `srotation` (rotation around s-axis) elements are patches element and define the following attribute:

angle
A *number* specifying the rotation angle around the axis of the element [rad]. (default: `0`).

5.22 ChangeRef

The `changeref` element is a patch element and defines the following attributes:

dx, dy, ds
A *number* specifying respectively x, y, s-translation of the reference frame [m]. (default: `0`)

dtheta, dphi, dpsi
A *number* specifying respectively y, $-x$, s-rotation of the reference frame applied in this order after any translation [rad]. (default: `0`)

5.23 ChangeDir

The `changedir` element is a patch element that reverses the direction of the sequence during the tracking.

5.24 ChangeNrj

The `changenrj` element is a patch element and defines the following attributes:

dnrj
A *number* specifying the change by δ_E of the *reference* beam energy [GeV]. The momenta of the particles or damaps belonging to the reference beam (i.e. not owning a beam) are updated, while other particles or damaps owning their beam are ignored. (default: `0`)

6 Flags

The `element` module exposes the following *object* flags through `MAD.element.flags` to use in conjunction with the methods `select` and `deselect`:[10]

none
All bits zero.

selected
Set if the element has been selected.

disabled
Set if the element has been disabled, e.g. for orbit correction.

observed
Set if the element has been selected for observation, e.g. for output to TFS table. The `$end` markers are selected for observation by default, and commands with the `observe` attribute set to `0` discard this flag and consider all elements as selected for observation.

implicit
Set if the element is implicit, like the temporary *implicit* drifts created on-the-fly by the `sequence` s-iterator with indexes at half integers. This flag is used by commands with the `implicit` attribute.

playout
Set if the element `angle` must be used by layout plot. This flag is useful to plot multiple sequence layouts around interaction points, like `lhcb1` and `lhcb2` around `IP1` and `IP5`.

7 Fringe fields

The `element` module exposes the following flags through `MAD.element.flags.fringe` to *control* the elements fringe fields through their attribute `fringe`, or to *restrict* the activated fringe fields with the commands attribute `fringe`:[11]

none
All bits zero.

bend
Control the element fringe fields for bending fields.

mult
Control the element fringe fields for multipolar fields up to `fringemax` order.

[10] Remember that flags are *not* inherited nor copied as they are qualifying the object itself.
[11] Those flags are *not* object flags, but fringe fields flags.

rfcav
Control the element fringe fields for rfcavity fields.

qsad
Control the element fringe fields for multipolar fields with extra terms for quadrupolar fields for compatibility with SAD.

comb
Control the element fringe fields for combined bending and multipolar fields.

combqs
Control the element fringe fields for combined bending and multipolar fields with extra terms for quadrupolar fields for compatibility with SAD.

The *element* `thick_element` provides a dozen of attributes to parametrize the aforementionned fringe fields. Note that in some future, part of these attributes may be grouped into a *mappable* to ensure a better consistency of their parametrization.

8 Sub-elements

An element can have thin or thick sub-elements stored in its *list* part, hence the length operator `#` returns the number of them. The attribute `sat` of sub-elements, i.e. read `sub-at`, is interpreted as their relative position from the entry of their enclosing main element, that is a fractional of its length. The positions of the sub-elements can be made absolute by dividing their `sat` attribute by the length of their main element using lambda expressions. The sub-elements are only considered and valid in the `drift_element` and `thick_element` kinds that implement the methods `:index_sat`, `:insert_sat`, `:remove_sat`, and `:replace_sat` to manage sub-elements from their `sat` attribute. The sequence method `:install` updates the `sat` attribute of the elements installed as sub-elements if the *logical* `elements.subelem` of the packed form is enabled, i.e. when the s-position determined by the `at`, `from` and `refpos` attributes falls inside a non-zero length element already installed in the sequence that is not an *implicit* drift. The physics of thick sub-elements will shield the physics of their enclosing main element along their length, unless they combine their attributes with those of their main element using lambda expressions to select some combined function physics.

9 Aperture

All the apertures are *mappable* defined by the following attributes in the tilted frame of an element, see the *track* command for details:

kind
A *string* specifying the aperture shape. (no default).

tilt
A *number* specifying the tilt angle of the aperture [rad]. (default: `0`).

xoff, yoff
A *number* specifying the transverse x, y-offset of the aperture [m]. (default: `0`).

maper
A *mappable* specifying a smaller aperture[12] than the `polygon` aperture to use before checking the

[12] It is the responsibility of the user to ensure that `maper` defines a smaller aperture than the polygon aperture.

polygon itself to speed up the test. The attributes `tilt`, `xoff` and `yoff` are ignored and superseded by the ones of the `polygon` aperture. (default: `nil`).

The supported aperture shapes are listed hereafter. The parameters defining the shapes are expected to be in the *list* part of the apertures and defines the top-right sector shape, except for the `polygon`:

square
A square shape with one parameter defining the side half-length. It is the default aperture check with limits set to 1.

rectangle
A rectangular shape with two parameters defining the x, y-half lengths (default: 1 [m]).

circle
A circular shape with one parameter defining the radius.

ellipse
A elliptical shape with two parameters defining the x, y-radii. (default: 1 [m]).

rectcircle
A rectangular shape intersected with a circular shape with three parameters defining the x, y-half lengths and the radius. (default: 1 [m]).

rectellipse
A rectangular shape intersected with an elliptical shape with four parameters defining the x, y-half lengths and the x, y-radii.

racetrack
A rectangular shape with corners rounded by an elliptical shape with four parameters defining the x, y-half lengths and the corners x, y-radii.

octagon
A rectangular shape with corners truncated by a triangular shape with four parameters defining the x, y-half lengths and the triangle x, y-side lengths. An octagon can model hexagon or diamond shapes by equating the triangle lengths to the rectangle half-lengths.

polygon
A polygonal shape defined by two vectors `vx` and `vy` holding the vertices coordinates. The polygon does not need to be convex, simple or closed, but in the latter case it will be closed automatically by joining the first and the last vertices.

bbox
A 6D bounding box with six parameters defining the upper limits of the absolute values of the six coordinates.

The following example defines new classes with three different aperture definitions:

```
local quadrupole in MAD.element
local mq = quadrupole 'mq' { l=1,                                   -- new class
  aperture = { kind='racetrack',
               tilt=pi/2, xoff=1e-3, yoff=5e-4,                     -- attributes
               0.06,0.06,0.01,0.01 }                                -- parameters
}
local mqdiam = quadrupole 'mqdiam' { l=1,                           -- new class
  aperture = { kind='octagon', xoff=1e-3, yoff=1e-3,                -- attributes
```

(continues on next page)

(continued from previous page)

```
                0.06,0.04,0.06,0.04 }                              -- parameters
}
local mqpoly = quadrupole 'mqpoly' { l=1,                          -- new class
  aperture = { kind='polygon', tilt=pi/2, xoff=1e-3, yoff=1e-3,    -- attributes
               vx=vector{0.05, ...}, vy=vector{0, ...},            -- parameters
               aper={kind='circle', 0.05}                          -- 2nd aperture
}
```

10 Misalignment

The misalignments are *mappable* defined at the entry of an element by the following attributes, see the *track* command for details:

dx, dy, ds
: A *number* specifying the x, y, s-displacement at the element entry [m], see Fig. 6.3 and Fig. 6.4 . (default: `0`).

dtheta
: A *number* specifying the y-rotation angle (azimuthal) at the element entry [rad], see Fig. 6.3. (default: `0`).

dphi
: A *number* specifying the $-x$-rotation angle (elevation) at the entry of the element [rad], see Fig. 6.5 . (default: `0`).

dpsi
: A *number* specifying the s-rotation angle (roll) at the element entry [rad], see Fig. 6.5 . (default: `0`).

Two kinds of misalignments are available for an element and summed beforehand:

- The *absolute* misalignments of the element versus its local reference frame, and specified by its `misalign` attribute. These misalignments are always considered.
- The *relative* misalignments of the element versus a given sequence, and specified by the *method* `:misalign` of `sequence`. These misalignments can be considered or not depending of command settings.

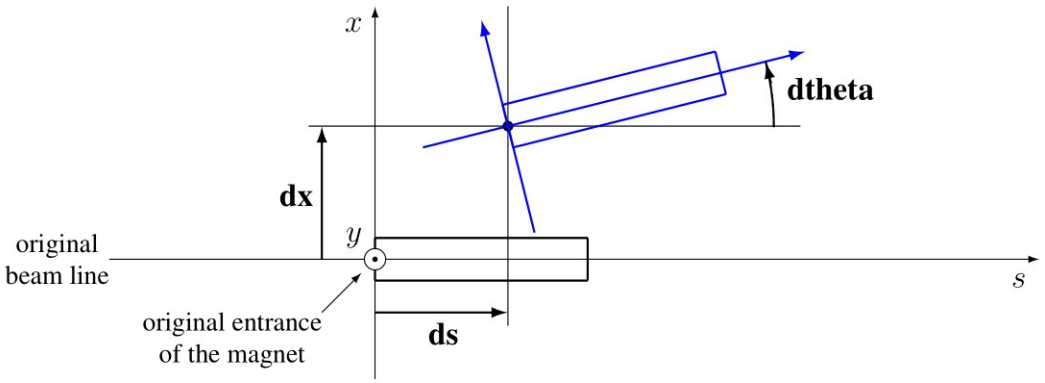

Figure6.3: Displacements in the (x, s) plane.

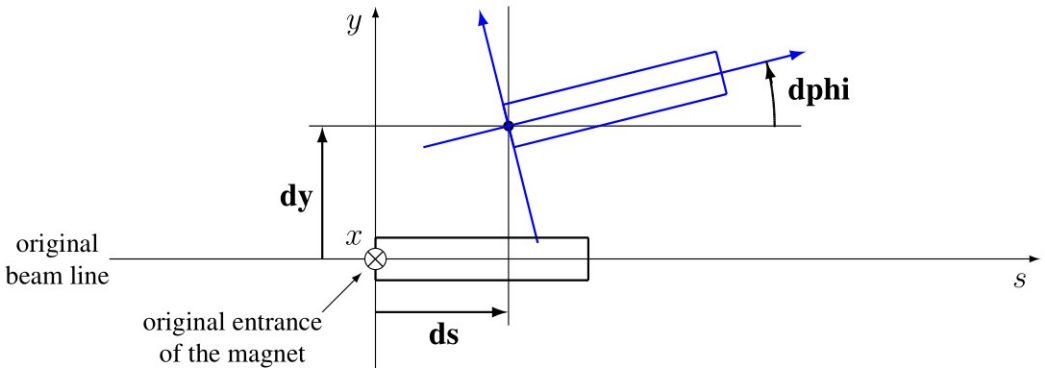

Figure6.4: Displacements in the (y, s) plane.

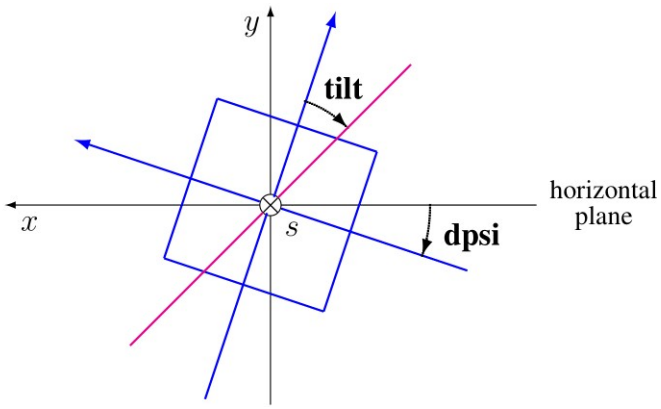

Figure6.5: Displacements in the (x, y) plane.

Chapter 7. 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 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 `"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* for details. (default: `nil` ≡ `"centre"`).

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

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*:

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

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

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

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 `: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 mtable 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

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

determined by `:foreach(filt_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 *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 *miscellaneous* 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]

update

A *method* `()` returning the sequence itself after recomputing the positions of all elements.

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

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

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.

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`.

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.

3 Metamethods

The `sequence` object provides the following metamethods:

__len
A *metamethod* () 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 *metamethod* `(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`.
2. Other *key* types are interpreted as *object* attributes subject to object model lookup.
3. 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]
4. Otherwise returns `nil`.

__newindex
A *metamethod* `(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 *metamethod* () called by the constructor to compute the elements positions.[5]

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

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

__sequ A unique private *reference* that characterizes sequences.

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' {
```

(continues on next page)

[4] An *iterable* supports the length operator `#`, the indexing operator `[]` and generic `for` loops with `ipairs`.
[5] MAD-NG does not have a MAD-X like `"USE"` command to finalize this computation.

(continued from previous page)

```
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
```

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.

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}`.

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.

8 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="^[^$]"}
-- display: all elements except $start and $end markers
```

(continues on next page)

(continued from previous page)

```
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="#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
```

9 Examples

9.1 FODO cell

The following example shows how to build a very simple FODO cell and an arc made of 10 FODO cells.

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

9.2 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' {}
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)
```

9.3 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
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
```

9.4 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()
```

Display:

```
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
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
```

(continues on next page)

(continued from previous page)

```
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
```

Chapter 8. MTables

The MAD Tables (MTables) — also named Table File System (TFS) — are objects convenient to store, read and write a large amount of heterogeneous information organized as columns and header. The MTables are also containers that provide fast access to their rows, columns, and cells by referring to their indexes, or some values of the designated reference column, or by running iterators constrained with ranges and predicates.

The `mtable` object is the *root object* of the TFS tables that store information relative to tables.

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

1 Attributes

The `mtable` object provides the following attributes:

type
: A *string* specifying the type of the mtable (often) set to the name of the command that created it, like `survey`, `track` or `twiss`. (default: `'user'`).

title
: A *string* specifying the title of the mtable (often) set to the attribute `title` of the command that created it. (default: `'no-title'`).

origin
: A *string* specifying the origin of the mtable. (default: `"MAD version os arch"`).

date
: A *string* specifying the date of creation of the mtable. (default: `"day/month/year"`).

time
: A *string* specifying the time of creation of the mtable. (default: `"hour:min:sec"`).

refcol
: A *string* specifying the name of the reference column used to build the dictionary of the mtable, and to mangle values with counts. (default: `nil`).

header
: A *list* specifying the augmented attributes names (and their order) used by default for the header when writing the mtable to files. Augmented meaning that the *list* is concatenated to the *list* held by the parent mtable during initialization. (default: `{'name', 'type', 'title', 'origin', 'date', 'time', 'refcol'}`).

column
: A *list* specifying the augmented columns names (and their order) used by default for the columns when writing the mtable to files. Augmented meaning that the *list* is concatenated to the *list* held by the parent mtable during initialization. (default: `nil`).

novector
: A *logical* specifying to not convert (`novector == true`) columns containing only numbers to vectors during the insertion of the second row. The attribute can also be a *list* specifying the columns names to remove from the specialization. If the *list* is empty or `novector ~= true`, all numeric columns

will be converted to vectors, and support all methods and operations from the *linear algebra* module. (default: `nil`).

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

reserve
A *number* specifying an estimate of the maximum number of rows stored in the mtable. If the value is underestimated, the mtable will still expand on need. (default: `8`).

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

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

__seq
A *sequence* attached to the mtable by the `survey` and `track` commands and used by the methods receiving a *reference* to an element as argument. (default: `nil`).

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

2 Methods

The `mtable` object provides the following methods:

nrow
A *method* `()` returning the *number* of rows in the mtable.

ncol
A *method* `()` returning the *number* of columns in the mtable.

ngen
A *method* `()` returning the *number* of columns generators in the mtable. The *number* of columns with data is given by `:ncol() - :ngen()`.

colname
A *method* `(idx)` returning the *string* name of the `idx`-th column in the mtable or `nil`.

colnames
A *method* `([lst])` returning the *list* `lst` (default: `{}`) filled with all the columns names of the mtable.

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

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

index_of
A *method* `(a, [ref], [dir])` returning a *number* corresponding to the positive index of the row

determined by the first argument or `nil`. If `a` is a *number* (or a *string* representing a *number*), it is interpreted as the index of the row and returned as a second *number*. If `a` is a *string*, it is interpreted as the (mangled) *value* of the row in the reference column as returned by `:name_of`. Finally, `a` can be a *reference* to an element to search for **if** the mtable has both, an attached sequence, and a column named `'eidx'` mapping the indexes of the elements to the attached sequence.[1] The argument `ref` (default: `nil`) specifies the reference row determined by `:index_of(ref)` to use for relative indexes, for decoding mangled values with relative counts, or as the reference row 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), which correspond respectively to the rounding methods `ceil`, `floor` and `round` from the lua math module.

range_of

A *method* `([rng], [ref], [dir])` returning three *number*s corresponding to the positive indexes *start* and *end* of the range and its direction *dir* (default: `1`), 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 *start* and *end*, both determined by `:index_of`. If `rng` is a *string* containing the separator `'/'`, it is split in two *string*s 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`. The arguments `ref` and `dir` 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])`.

get

A *method* `(row, col, [cnt])` returning the *value* stored in the mtable at the cell `(row,col)`, or `nil`. If `row` is a not a row index determined by `:index(row)`, it is interpreted as a (mangled) *value* to search in the reference column, taking into account the count `cnt` (default: `1`). If `col` is not a column index, it is interpreted as a column name.

set

A *method* `(row, col, val, [cnt])` returning the mtable itself after updating the cell `(row,col)` to the value `val`, or raising an error if the cell does not exist. If `row` is a not a row index determined by `:index(row)`, it is interpreted as a (mangled) *value* to search in the reference column, taking into account the count `cnt` (default: `1`). If `col` is not a column index, it is interpreted as a column name.

getcol

A *method* `(col)` returning the column `col`, or `nil`. If `col` is not a column index, it is interpreted as a column name.

setcol

A *method* `(col, val)` returning the mtable itself after updating the column `col` with the values of `val`, or raising an error if the column does not exist. If `col` is not a column index, it is interpreted as a column name. If the column is a generator, so must be `val` or an error will be raised. If the column is not a generator and `val` is a *callable* `(ri)`, it will be invoked with the row index `ri` as its sole argument, using its returned value to update the column cell. Otherwise `val` must be an *iterable* or an error will be raised. If the column is already a specialized *vector*, the *iterable* must provide enough

[1] These information are usually provided by the command creating the `mtable`, like `survey` and `track`.

numbers to fill it entirely as `nil` is not a valid value.

inscol

A *method* `([ref], col, val, [nvec])` returning the mtable itself after inserting the column data `val` with the *string* name `col` at index `ref` (default: `:ncol()+1`). If `ref` is not a column index, it is interpreted as a column name. If `val` is a *callable* `(ri)`, it will be added as a column generator. Otherwise `val` must be an *iterable* or an error will be raised. The *iterable* will used to fill the new column that will be specialized to a *vector* if its first value is a *number* and `nvec ~= true` (default: `nil`).

addcol

A *method* `(col, val, [nvec])` equivalent to `:inscol(nil, col, val, [nvec])`.

remcol

A *method* `(col)` returning the mtable itself after removing the column `col`, or raising an error if the column does not exist. If `col` is not a column index, it is interpreted as a column name.

rencol

A *method* `(col, new)` returning the mtable itself after renaming the column `col` to the *string* `new`, or raising an error if the column does not exist. If `col` is not a column index, it is interpreted as a column name.

getrow

A *method* `(row, [ref])` returning the *mappable* (proxy) of the row determined by the method `:index_of(row, [ref])`, or `nil`.

setrow

A *method* `(row, val, [ref])` returning the mtable itself after updating the row at index determined by `:index_of(row, [ref])` using the values provided by the *mappable* `val`, which can be a *list* iterated as pairs of (*index*, *value*) or a *set* iterated as pairs of (*key*, *value*) with *key* being the column names, or a combination of the two. An error is raised if the column does not exist.

insrow

A *method* `(row, val, [ref])` returning the mtable itself after inserting a new row at index determined by `:index_of(row, [ref])` and filled with the values provided by the *mappable* `val`, which can be a *list* iterated as pairs of (*index*, *value*) or a *set* iterated as pairs of (*key*, *value*) with *key* being the column names or a combination of the two.

addrow

A *method* `(val)` equivalent to `:insrow(#self+1, val)`.

remrow

A *method* `(row, [ref])` returning the mtable itself after removing the row determined by the method `:index_of(row, [ref])`, or raising an error if the row does not exist.

swprow

A *method* `(row1, row2, [ref1], [ref2])` returning the mtable itself after swapping the content of the rows, both determined by the method `:index_of(row, [ref])`, or raising an error if one of the row does not exist.

clrrow

A *method* `(row, [ref])` returning the mtable itself after clearing the row determined by the method `:index_of(row, [ref])`, or raising an error if the row does not exist; where clearing the row means to set *vector* value to `0` and `nil` otherwise.

clear

A *method* `()` returning the mtable itself after clearing all the rows, i.e. `#self == 0`, with an opportunity for new columns specialization.

iter

A *method* `([rng], [ntrn], [dir])` returning an iterator over the mtable rows. The optional range is determined by `: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 mtable 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 rows the index and the row *mappable* (proxy).

foreach

A *method* `(act, [rng], [sel], [not])` returning the mtable itself after applying the action `act` on the selected rows. 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* `(row, idx)` applied to a row passed as first argument and its index as second argument. The optional range is used to generate the loop iterator `:iter([rng])`. The optional selector `sel` is a *callable* `(row, idx)` predicate selecting eligible rows for the action from the row itself passed as first argument and its index as second argument. The selector `sel` can be specified in other ways, see *row 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* `([rng], [sel], [not])` returning the mtable itself after selecting the rows using `:foreach(sel_act, [rng], [sel], [not])`. By default mtable have all their rows deselected, the selection being stored as *boolean* in the column at index `0` and named `is_selected`.

deselect

A *method* `([rng], [sel], [not])` returning the mtable itself after deselecting the rows using `:foreach(desel_act, [rng], [sel], [not])`. By default mtable have all their rows deselected, the selection being stored as *boolean* in the column at index `0` and named `is_selected`.

filter

A *method* `([rng], [sel], [not])` returning a *list* containing the positive indexes of the rows determined by `:foreach(filt_act, [rng], [sel], [not])`, and its size.

insert

A *method* `(row, [rng], [sel])` returning the mtable itself after inserting the rows in the *list* `row` at the indexes determined by `:filter([rng], [sel], true)`. If the arguments are passed in the packed form, the extra attribute `rows` will be used as a replacement for the argument `row`, and if the attribute `where="after"` is defined then the rows will be inserted after the selected indexes. The insertion scheme depends on the number R of rows in the *list* `row` versus the number S of rows selected by `:filter`; 1×1 (one row inserted at one index), $R \times 1$ (R rows inserted at one index), $1 \times S$ (one row inserted at S indexes) and $R \times S$ (R rows inserted at S indexes). Hence, the insertion schemes insert respectively 1, R, S, and $\min(R, S)$ rows.

remove

A *method* `([rng], [sel])` returning the mtable itself after removing the rows determined by `:filter([rng], [sel], true)`.

sort

A *method* `(cmp, [rng], [sel])` returning the mtable itself after sorting the rows at the indexes

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

determined by `:filter([rng], [sel], true)` using the ordering *callable* `cmp(row1, row2)`. The arguments `row1` and `row2` are *mappable* (proxies) referring to the current rows being compared and providing access to the columns values for the comparison.[3] The argument `cmp` can be specified in a compact ordering form as a *string* that will be converted to an ordering *callable* by the function `str2cmp` from the *utility* module. For example, the *string* "-y,x" will be converted by the method to the following *lambda* `\r1,r2 -> r1.y > r2.y or r1.y == r2.y and r1.x < r2.x`, where `y` and `x` are the columns used to sort the mtable in descending (-) and ascending (+) order respectively. The compact ordering form is not limited in the number of columns and avoids making mistakes in the comparison logic when many columns are involved.

cycle
A *method* `(a)` returning the mtable 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 mtable like `:insert`, `:remove` or `:sort`.

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

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

set_readonly
Set the mtable as read-only, including the columns and the rows proxies.

read
A *method* `([filname])` returning a new instance of `self` filled with the data read from the file determined by `openfile(filename, 'r', {'.tfs','.txt','.dat'})` from the *utility* module. This method can read columns containing the data types *nil*, *boolean*, *number*, *complex number*, (numerical) *range*, and (quoted) *string*. The header can also contain tables saved as *string* and decoded with *function* `str2tbl` from the *utility* module.

write
A *method* `([filname], [clst], [hlst], [rsel])` returning the mtable itself after writing its content to the file determined by `openfile(filename, 'w', {'.tfs', '.txt', '.dat'})` from the *utility* module. The columns to write and their order is determined by `clst` or `self.column` (default: `nil` ≡ all columns). The attributes to write in the header and their order is determined by `hlst` or `self.header`. The *logical* `rsel` indicates to save all rows or only rows selected by the `:select` method (`rsel == true`). This method can write columns containing the data types *nil*, *boolean*, *number*, *complex number*, (numerical) *range*, and (quoted) *string*. The header can also contain tables saved as *string* and encoded with *function* `tbl2str` from the *utility* module.

print
A *method* `([clst], [hlst], [rsel])` equivalent to `:write(nil, [clst], [hlst], [rsel])`.

save_sel
A *method* `([sel])` saving the rows selection to the optional *iterable* `sel` (default: `{}`) and return it.

restore_sel
A *method* `(sel)` restoring the rows selection from the *iterable* `sel`. The indexes of `sel` must match

[3] A *mappable* supports the length operator `#`, the indexing operator `[]`, and generic `for` loops with `pairs`.

the indexes of the rows in the mtable.

make_dict
A *method* `([col])` returning the mtable itself after building the rows dictionnary from the values of the reference column determined by `col` (default: `refcol`) for fast row access. If `col` is not a column index, it is interpreted as a column name except for the special name `'none'` that disables the rows dictionnary and reset `refcol` to `nil`.

check_mtbl
A *method* `()` checking the integrity of the mtable and its dictionary (if any), for debugging purpose only.

3 Metamethods

The `mtable` object provides the following metamethods:

__len
A *metamethod* `()` called by the length operator `#` to return the number of rows in the mtable.

__add
A *metamethod* `(val)` called by the plus operator + returning the mtable itself after appending the row `val` at its end, similiar to the `:addrow` method.

__index
A *metamethod* `(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 a row index and returns an *iterable* on the row (proxy) or `nil`.
2. Other *key* types are interpreted as *object* attributes subject to object model lookup.
3. If the *value* associated with *key* is `nil`, then *key* is interpreted as a column name and returns the column if it exists, otherwise...
4. If *key* is not a column name, then *key* is interpreted as a value in the reference column and returns either an *iterable* on the row (proxy) determined by this value or an *iterable* on the rows (proxies) holding this non-unique value.[4]
5. Otherwise returns `nil`.

__newindex
A *metamethod* `(key, val)` called by the assignment operator `[key]=val` to create new attributes for the pairs (*key*, *value*). If *key* is a *number* or a value specifying a row in the reference column or a *string* specifying a column name, the following error is raised:

```
"invalid mtable write access (use 'set' methods)"
```

__init
A *metamethod* `()` called by the constructor to build the mtable from the column names stored in its *list* part and some attributes, like `owner`, `reserve` and `novector`.

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

[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:

__mtbl
: A unique private *reference* that characterizes mtables.

4 MTables creation

During its creation as an *object*, an mtable can defined its attributes as any object, and the *list* of its column names, which will be cleared after its initialization. Any column name in the *list* that is enclosed by braces is designated as the refererence column for the dictionnary that provides fast row indexing, and the attribute `refcol` is set accordingly.

Some attributes are considered during the creation by the *metamethod* `__init`, like `owner`, `reserve` and `novector`, and some others are initialized with defined values like `type`, `title`, `origin`, `date`, `time`, and `refcol`. The attributes `header` and `column` are concatenated with the the parent ones to build incrementing *list* of attributes names and columns names used by default when writing the mtable to files, and these lists are not provided as arguments.

The following example shows how to create a mtable form a *list* of column names add rows:

```
local mtable in MAD
local tbl = mtable 'mytable' {

   {'name'}, 'x', 'y' } -- column 'name' is the refcol
  + { 'p11', 1.1, 1.2 }
  + { 'p12', 2.1, 2.2 }
  + { 'p13', 2.1, 3.2 }
  + { 'p11', 3.1, 4.2 }
print(tbl.name, tbl.refcol, tbl:getcol'name')
-- display: mytable  name   mtable reference column: 0x010154cd10
```

Pitfall: When a column is named `'name'`, it must be explicitly accessed, e.g. with the `:getcol` method, as the indexing operator `[]` gives the precedence to object's attributes and methods. Hence, `tbl.name` returns the table name `'mytable'`, not the column `'name'`.

5 Rows selections

The row selection in mtable use predicates in combination with iterators. The mtable iterator manages the range of rows where to apply the selection, while the predicate says if a row 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:

selected
: A *boolean* compared to the rows selection stored in column `'is_selected'`.

pattern
: A *string* interpreted as a pattern to match the *string* in the reference column, which must exist, using `string.match` from the standard library, see Lua 5.2 §6.4 for details. If the reference column does not exist, it can be built using the `make_dict` method.

list
: An *iterable* interpreted as a *list* used to build a *set* and select the rows by their name, i.e. the built predicate will use `tbl[row.name]` as a *logical*, meaning that column `name` must exists. An alternate column name can be provided through the key `colname`, i.e. used as `tbl[row[colname]]`. 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 rows by their name, i.e. the built predicate will use `tbl[row.name]` as a *logical*, meaning that column `name` must exists. 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.

kind
: An *iterable* interpreted as a *list* used to build a *set* and select the rows by their kind, i.e. the built predicate will use `tbl[row.kind]` as a *logical*, meaning that column `kind` must exists. If the *iterable* is a single item, e.g. a *string*, it will be converted first to a *list*. This case is equivalent to `list` with `colname='kind'`.

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

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, `tbl:foreach(act, "^MB")` is equivalent to `tbl:foreach{action=act, pattern="^MB"}`.

6 Indexes, names and counts

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

If a row exists but its *value* is not unique in the reference column, an *iterable* is returned. An *iterable* supports the length `#` operator to retrieve the number of rows with the same *value*, the indexing operator `[]` waiting for a count n to retrieve the n-th row from the start with that *value*, 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.

Note: Compared to the sequence, the indexing operator `[]` and the method `:index_of` of the mtable always interprets a *number* as a (relative) row index. To find a row from a s-position [m] in the mtable if the column exists, use the functions `lsearch` or `bsearch` (if they are monotonic) from the *utility* module.

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

```
local mtable in MAD
local tbl = mtable { {'name'}, 'x', 'y' } -- column 'name' is the refcol
                   + { 'p11', 1.1, 1.2 }
                   + { 'p12', 2.1, 2.2 }
                   + { 'p13', 2.1, 3.2 }
                   + { 'p11', 3.1, 4.2 }
print(tbl[ 1].y) -- display: 1.2
print(tbl[-1].y) -- display: 4.2

print(#tbl.p11, tbl.p12.y, tbl.p11[2].y)            -- display: 2 2.2 4.2
for _,r in ipairs(tbl.p11) do io.write(r.x," ") end -- display: 1.1 3.1
for _,v in ipairs(tbl.p12) do io.write(v,  " ") end -- display: 'p12' 2.1 2.2

-- print name of point with name p11 in absolute and relative to p13.
print(tbl:name_of(4))       -- display: p11[2]  (2nd p11 from start)
print(tbl:name_of(1, -2))   -- display: p11{-1} (1st p11 before p13)
```

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

7 Iterators and ranges

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

The mtable iterators are created by the method `:iter`, based on the method `:range_of` as mentioned in its description 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 method `:foreach` uses the iterator returned by `:iter` with a range as its sole argument to loop over the rows where to apply the predicate before executing the action. The methods `:select`, `:deselect`, `:filter`, `:insert`, and `:remove` are all based directly or indirectly on the `:foreach` method. Hence, to iterate backward over a mtable range, these methods have to use either its *list* form or a numerical range. For example the invocation `tbl:foreach(\r -> print(r.name), {-2, 2, nil, -1})` will iterate backward over the entire mtable excluding the first and last rows, equivalently to the invocation `tbl:foreach(\r -> print(r.name), -2..2..-1)`.

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

```
local mtable in MAD
local tbl = mtable { {'name'}, 'x', 'y' }
                   + { 'p11', 1.1, 1.2 }
                   + { 'p12', 2.1, 2.2 }
                   + { 'p13', 2.1, 3.2 }
                   + { 'p11', 3.1, 4.2 }
```

(continues on next page)

(continued from previous page)

```
local act = \r -> print(r.name, r.y)
tbl:foreach(act, -2..2..-1)
-- display:  p13   3.2
!            p12   2.2
tbl:foreach(act, "p11[1]/p11[2]")
-- display:  p11   1.2
!            p12   2.2
!            p13   3.2
!            p11   4.2
tbl:foreach{action=act, range="p11[1]/p13"}
-- display:  p11   1.2
!            p12   2.2
!            p13   3.2
tbl:foreach{action=act, pattern="[^1]$"}
-- display:  p12   2.2
!            p13   3.2
local act = \r -> print(r.name, r.y, r.is_selected)
tbl:select{pattern="p.1"}:foreach{action=act, range="1/-1"}
-- display:  p11   1.2   true
!            p12   2.2   nil
!            p13   3.2   nil
!            p11   4.2   true
```

8 Examples

8.1 Creating a MTable

The following example shows how the `track` command, i.e. `self` hereafter, creates its MTable:

```
local header = { -- extra attributes to save in track headers
  'direction', 'observe', 'implicit', 'misalign', 'deltap', 'lost' }

local function make_mtable (self, range, nosave)
  local title, dir, observe, implicit, misalign, deltap, savemap in self
  local sequ, nrow = self.sequence, nosave and 0 or 16

  return mtable(sequ.name, { -- keep column order!
    type='track', title=title, header=header,
    direction=dir, observe=observe, implicit=implicit, misalign=misalign,
    deltap=deltap, lost=0, range=range, reserve=nrow, __seq=sequ,
    {'name'}, 'kind', 's', 'l', 'id', 'x', 'px', 'y', 'py', 't', 'pt',
    'slc', 'turn', 'tdir', 'eidx', 'status', savemap and '__map' or nil })
end
```

8.2 Extending a MTable

The following example shows how to extend the MTable created by a `twiss` command with the elements tilt, angle and integrated strengths from the attached sequence:

```
-- The prelude creating the sequence seq is omitted.
local tws = twiss { sequence=seq, method=4, cofind=true }

local is_integer in MAD.typeid
tws:addcol('angle', \ri => -- add angle column
      local idx = tws[ri].eidx
      return is_integer(idx) and tws.__seq[idx].angle or 0 end)
   :addcol('tilt', \ri => -- add tilt column
      local idx = tws[ri].eidx
      return is_integer(idx) and tws.__seq[idx].tilt or 0 end)

for i=1,6 do -- add kil and kisl columns
tws:addcol('k'..i-1..'l', \ri =>
      local idx = tws[ri].eidx
      if not is_integer(idx) then return 0 end -- implicit drift
      local elm = tws.__seq[idx]
      return (elm['k'..i-1] or 0)*elm.l + ((elm.knl or {})[i] or 0)
    end)
   :addcol('k'..i-1..'sl', \ri =>
      local idx = tws[ri].eidx
      if not is_integer(idx) then return 0 end -- implicit drift
      local elm = tws.__seq[idx]
      return (elm['k'..i-1..'s'] or 0)*elm.l + ((elm.ksl or {})[i] or 0)
    end)
end

local cols = {'name', 'kind', 's', 'l', 'angle', 'tilt',
    'x', 'px', 'y', 'py', 't', 'pt',
    'beta11', 'beta22', 'alfa11', 'alfa22', 'mu1', 'mu2', 'dx', 'ddx',
    'k1l', 'k2l', 'k3l', 'k4l', 'k1sl', 'k2sl', 'k3sl', 'k4sl'}

tws:write("twiss", cols) -- write header and columns to file twiss.tfs
```

Hopefully, the *physics* module provides the *function* `melmcol(mtbl, cols)` to achieve the same task easily:

```
-- The prelude creating the sequence seq is omitted.
local tws = twiss { sequence=seq, method=4, cofind=true }

-- Add element properties as columns
local melmcol in MAD.gphys
local melmcol(tws, {'angle', 'tilt', 'k1l' , 'k2l' , 'k3l' , 'k4l',
                                     'k1sl', 'k2sl', 'k3sl', 'k4sl'})
```

(continues on next page)

(continued from previous page)

```
-- write TFS table
tws:write("twiss", {
    'name', 'kind', 's', 'l', 'angle', 'tilt',
    'x', 'px', 'y', 'py', 't', 'pt',
    'beta11', 'beta22', 'alfa11', 'alfa22', 'mu1', 'mu2', 'dx', 'ddx',
    'k1l', 'k2l', 'k3l', 'k4l', 'k1sl', 'k2sl', 'k3sl', 'k4sl'})
```

Chapter 9. MADX

Part II

ELEMENTS & COMMANDS

Chapter 10. Survey

The survey command provides a simple interface to the *geometric* tracking code.[1] The geometric tracking can be used to place the elements of a sequence in the global reference system in Fig. 18.2.

Listing 10.1: Synopsis of the survey command with default setup.

```
mtbl, mflw [, eidx] = survey {
        sequence=sequ,  -- sequence (required)
        range=nil,      -- range of tracking (or sequence.range)
        dir=1,          -- s-direction of tracking (1 or -1)
        s0=0,           -- initial s-position offset [m]
        X0=0,           -- initial coordinates x, y, z [m]
        A0=0,           -- initial angles theta, phi, psi [rad] or matrix W0
        nturn=1,        -- number of turns to track
        nstep=-1,       -- number of elements to track
        nslice=1,       -- number of slices (or weights) for each element
        implicit=false, -- slice implicit elements too (e.g. plots)
        misalign=false, -- consider misalignment
        save=true,      -- create mtable and save results
        title=nil,      -- title of mtable (default seq.name)
        observe=0,      -- save only in observed elements (every n turns)
        savesel=fnil,   -- save selector (predicate)
        savemap=false,  -- save the orientation matrix W in the column __map
        atentry=fnil,   -- action called when entering an element
        atslice=fnil,   -- action called after each element slices
        atexit=fnil,    -- action called when exiting an element
        atsave=fnil,    -- action called when saving in mtable
        atdebug=fnil,   -- action called when debugging the element maps
        info=nil,       -- information level (output on terminal)
        debug=nil,      -- debug information level (output on terminal)
        usrdef=nil,     -- user defined data attached to the mflow
        mflow=nil,      -- mflow, exclusive with other attributes except nstep
}
```

1 Command synopsis

The survey command format is summarized in Listing 10.1, including the default setup of the attributes. The survey command supports the following attributes:

sequence

The *sequence* to survey. (no default, required).

Example: `sequence = lhcb1`.

range

A *range* specifying the span of the sequence survey. If no range is provided, the command looks for a

[1] MAD-NG implements only two tracking codes denominated the *geometric* and *dynamic* tracking

range attached to the sequence, i.e. the attribute . (default: `nil`).

Example: `range = "S.DS.L8.B1/E.DS.R8.B1"`.

dir

The s-direction of the tracking: `1` forward, `-1` backward. (default: `1`).

Example: `dir = -1`.

s0

A *number* specifying the initial s-position offset. (default: `0` [m]).

Example: `s0 = 5000`.

X0

A *mappable* specifying the initial coordinates `{x,y,z}`. (default: `0` [m]).

Example: `X0 = { x=100, y=-50 }`

A0

A *mappable* specifying the initial angles `theta`, `phi` and `psi` or an orientation *matrix* `W0`.[2] (default: `0` [rad]).

Example: `A0 = { theta=deg2rad(30) }`

nturn

A *number* specifying the number of turn to track. (default: `1`).

Example: `nturn = 2`.

nstep

A *number* specifying the number of element to track. A negative value will track all elements. (default: `-1`).

Example: `nstep = 1`.

nslice

A *number* specifying the number of slices or an *iterable* of increasing relative positions or a *callable* `(elm, mflw, lw)` returning one of the two previous kind of positions to track in the elements. The arguments of the callable are in order, the current element, the tracked map flow, and the length weight of the step. This attribute can be locally overridden by the element. (default: `1`).

Example: `nslice = 5`.

implicit

A *logical* indicating that implicit elements must be sliced too, e.g. for smooth plotting. (default: `false`).

Example: `implicit = true`.

misalign

A *logical* indicating that misalignment must be considered. (default: `false`).

Example: `implicit = true`.

save

A *logical* specifying to create a *mtable* and record tracking information at the observation points. The `save` attribute can also be a *string* specifying saving positions in the observed elements: `"atentry"`, `"atslice"`, `"atexit"` (i.e. `true`), `"atbound"` (i.e. entry and exit), `"atbody"` (i.e. slices and exit) and `"atall"`. (default: `true`).

Example: `save = false`.

[2] An orientation matrix can be obtained from the 3 angles with `W=matrix(3):rotzxy(- phi,theta,psi)`

title

A *string* specifying the title of the *mtable*. If no title is provided, the command looks for the name of the sequence, i.e. the attribute `seq.name`. (default: `nil`).

Example: `title = "Survey around IP5"`.

observe

A *number* specifying the observation points to consider for recording the tracking information. A zero value will consider all elements, while a positive value will consider selected elements only, checked with method `:is_observed`, every `observe` > 0 turns. (default: `0`).

Example: `observe = 1`.

savesel

A *callable* `(elm, mflw, lw, islc)` acting as a predicate on selected elements for observation, i.e. the element is discarded if the predicate returns `false`. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`)

Example: `savesel = \e -> mylist[e.name] ~= nil`.

savemap

A *logical* indicating to save the orientation matrix `W` in the column `__map` of the *mtable*. (default: `false`).

Example: `savemap = true`.

atentry

A *callable* `(elm, mflw, 0, -1)` invoked at element entry. The arguments are in order, the current element, the tracked map flow, zero length and the slice index `-1`. (default: `fnil`).

Example: `atentry = myaction`.

atslice

A *callable* `(elm, mflw, lw, islc)` invoked at element slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`).

Example: `atslice = myaction`.

atexit

A *callable* `(elm, mflw, 0, -2)` invoked at element exit. The arguments are in order, the current element, the tracked map flow, zero length and the slice index `-2`. (default: `fnil`).

Example: `atexit = myaction`.

atsave

A *callable* `(elm, mflw, lw, islc)` invoked at element saving steps, by default at exit. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`).

Example: `atsave = myaction`.

atdebug

A *callable* `(elm, mflw, lw, [msg], [...])` invoked at the entry and exit of element maps during the integration steps, i.e. within the slices. The arguments are in order, the current element, the tracked map flow, the length weight of the integration step and a *string* specifying a debugging message, e.g. `"map_name:0"` for entry and `":1"` for exit. If the level `debug` ≥ 4 and `atdebug` is not specified, the default *function* `mdump` is used. In some cases, extra arguments could be passed to the method. (default: `fnil`).

Example: `atdebug = myaction` .

info

A *number* specifying the information level to control the verbosity of the output on the console. (default: `nil`).

Example: `info = 2`.

debug

A *number* specifying the debug level to perform extra assertions and to control the verbosity of the output on the console. (default: `nil`).

Example: `debug = 2`.

usrdef

Any user defined data that will be attached to the tracked map flow, which is internally passed to the elements method `:survey` and to their underlying maps. (default: `nil`).

Example: `usrdef = { myvar=somevalue }`.

mflow

A *mflow* containing the current state of a `survey` command. If a map flow is provided, all attributes are discarded except `nstep`, `info` and `debug`, as the command was already set up upon its creation. (default: `nil`).

Example: `mflow = mflow0`.

The `survey` command returns the following objects in this order:

mtbl

A *mtable* corresponding to the TFS table of the `survey` command.

mflw

A *mflow* corresponding to the map flow of the `survey` command.

eidx

An optional *number* corresponding to the last surveyed element index in the sequence when `nstep` was specified and stopped the command before the end of the `range`.

2 Survey mtable

The `survey` command returns a *mtable* where the information described hereafter is the default list of fields written to the TFS files.[3]

The header of the *mtable* contains the fields in the default order:

name

The name of the command that created the *mtable*, e.g. `"survey"`.

type

The type of the *mtable*, i.e. `"survey"`.

title

The value of the command attribute `title`.

origin

The origin of the application that created the *mtable*, e.g. `"MAD 1.0.0 OSX 64"`.

date

The date of the creation of the *mtable*, e.g. `"27/05/20"`.

[3] The output of mtable in TFS files can be fully customized by the user.

time
The time of the creation of the *mtable*, e.g. `"19:18:36"`.

refcol
The reference *column* for the *mtable* dictionnary, e.g. `"name"`.

direction
The value of the command attribute `dir`.

observe
The value of the command attribute `observe`.

implicit
The value of the command attribute `implicit`.

misalign
The value of the command attribute `misalign`.

range
The value of the command attribute `range`.[4]

__seq
The *sequence* from the command attribute `sequence`.[5]

The core of the *mtable* contains the columns in the default order:

name
The name of the element.

kind
The kind of the element.

s
The s-position at the end of the element slice.

l
The length from the start of the element to the end of the element slice.

angle
The angle from the start of the element to the end of the element slice.

tilt
The tilt of the element.

x
The global coordinate x at the s-position.

y
The global coordinate y at the s-position.

z
The global coordinate z at the s-position.

theta
The global angle θ at the s-position.

phi
The global angle ϕ at the s-position.

[4] This field is not saved in the TFS table by default.
[5] Fields and columns starting with two underscores are protected data and never saved to TFS files.

psi
The global angle ψ at the s-position.

slc
The slice number ranging from `-2` to `nslice`.

turn
The turn number.

tdir
The t-direction of the tracking in the element.

eidx
The index of the element in the sequence.

__map
The orientation *matrix* at the s-position.[Page 97, 5]

3 Geometrical tracking

Fig. 10.1 presents the scheme of the geometrical tracking through an element sliced with `nslice=3`. The actions `atentry` (index `-1`), `atslice` (indexes `0..3`), and `atexit` (index `-2`) are reversed between the forward tracking (`dir=1` with increasing s-position) and the backward tracking (`dir=-1` with decreasing s-position). By default, the action `atsave` is attached to the exit slice, and hence it is also reversed in the backward tracking.

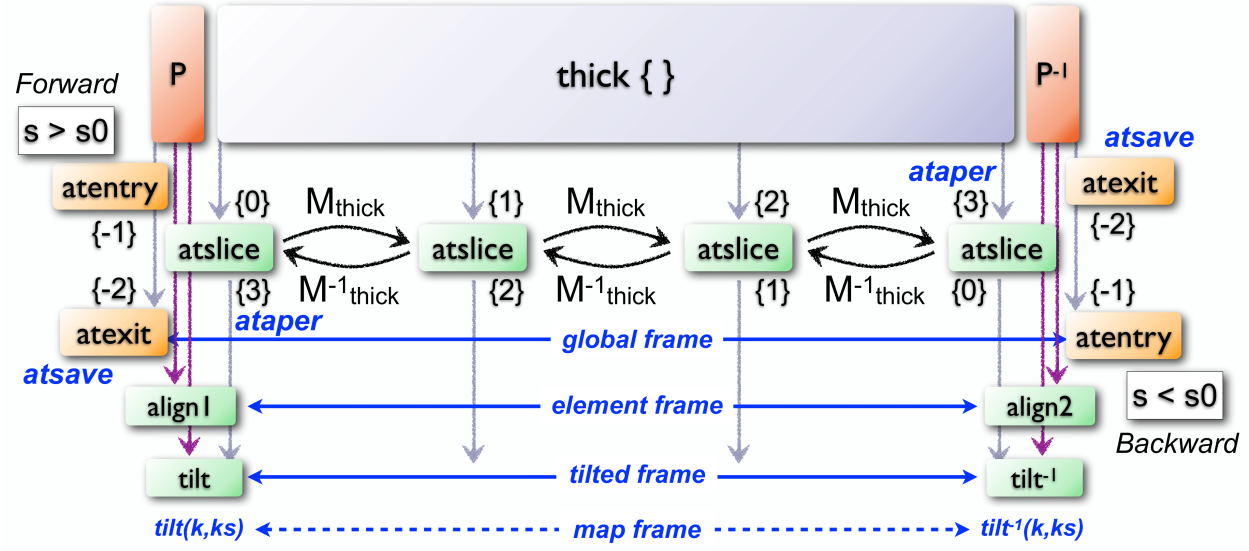

Figure10.1: Geometrical tracking with slices.

3.1 Slicing

The slicing can take three different forms:

- A *number* of the form `nslice`=N that specifies the number of slices with indexes `0..N`. This defines a uniform slicing with slice length $l_{\text{slice}} = l_{\text{elem}}/N$.
- An *iterable* of the form `nslice={lw_1,lw_2,..,lw_N}` with $\sum_i lw_i = 1$ that specifies the fraction of length of each slice with indexes `0..N` where N =`#nslice`. This defines a non-uniform slicing with a slice length of $l_i = lw_i \times l_{\text{elem}}$.
- A *callable* `(elm, mflw, lw)` returning one of the two previous forms of slicing. The arguments are in order, the current element, the tracked map flow, and the length weight of the step, which should allow to return a user-defined element-specific slicing.

The surrounding `P` and $\texttt{P}^{-1}$ maps represent the patches applied around the body of the element to change the frames, after the `atentry` and before the `atexit` actions:

- The misalignment of the element to move from the *global frame* to the *element frame* if the command attribute `misalign` is set to `true`.
- The tilt of the element to move from the element frame to the *titled frame* if the element attribute `tilt` is non-zero. The `atslice` actions take place in this frame.

These patches do not change the global frame per se, but they may affect the way that other components change the global frame, e.g. the tilt combined with the angle of a bending element.

3.2 Sub-elements

The `survey` command takes sub-elements into account, mainly for compatibility with the `track` command. In this case, the slicing specification is taken between sub-elements, e.g. 3 slices with 2 sub-elements gives a final count of 9 slices. It is possible to adjust the number of slices between sub-elements with the third form of slicing specifier, i.e. by using a callable where the length weight argument is between the current (or the end of the element) and the last sub-elements (or the start of the element).

4 Examples

Chapter 11. Track

The `track` command provides a simple interface to the *dynamic* tracking code.[1] The dynamic tracking can be used to track the particles in the *local reference system* while running through the elements of a sequence. The particles coordinates can be expressed in the *global reference system* by changing from the local to the global frames using the information delivered by the *survey* command.

Listing 11.1: Synopsis of the `track` command with default setup.

```
mtbl, mflw [, eidx] = track {
 sequence=sequ, -- sequence (required)
 beam=nil,      -- beam (or sequence.beam, required)
 range=nil,     -- range of tracking (or sequence.range)
 dir=1,         -- s-direction of tracking (1 or -1)
 s0=0,          -- initial s-position offset [m]
 X0=0,          -- initial coordinates (or damap(s), or beta block(s))
 O0=0,          -- initial coordinates of reference orbit
 deltap=nil,    -- initial deltap(s)
 nturn=1,       -- number of turns to track
 nstep=-1,      -- number of elements to track
 nslice=1,      -- number of slices (or weights) for each element
 mapdef=false,          -- setup for damap (or list of, true => {})
 method=2,      -- method or order for integration (1 to 8)
 model='TKT',   -- model for integration ('DKD' or 'TKT')
 ptcmodel=nil,          -- use strict PTC thick model (override option)
 implicit=false,        -- slice implicit elements too (e.g. plots)
 misalign=false,        -- consider misalignment
 fringe=true,   -- enable fringe fields (see element.flags.fringe)
 radiate=false,         -- radiate at slices
 totalpath=false,       -- variable 't' is the totalpath
 save=true,     -- create mtable and save results
 title=nil,     -- title of mtable (default seq.name)
 observe=1,     -- save only in observed elements (every n turns)
 savesel=fnil,          -- save selector (predicate)
 savemap=false,         -- save damap in the column __map
 atentry=fnil,          -- action called when entering an element
 atslice=fnil,          -- action called after each element slices
 atexit=fnil,   -- action called when exiting an element
 ataper=fnil,   -- action called when checking for aperture
 atsave=fnil,   -- action called when saving in mtable
 atdebug=fnil,  -- action called when debugging the element maps
 info=nil,      -- information level (output on terminal)
 debug=nil,     -- debug information level (output on terminal)
 usrdef=nil,    -- user defined data attached to the mflow
 mflow=nil,     -- mflow, exclusive with other attributes except nstep
```

(continues on next page)

[1] MAD-NG implements only two tracking codes denominated the *geometric* and the *dynamic* tracking.

(continued from previous page)

```
}
```

1 Command synopsis

The `track` command format is summarized in Listing 11.1, including the default setup of the attributes.
The `track` command supports the following attributes:

sequence
: The *sequence* to track. (no default, required).
Example: `sequence = lhcb1`.

beam
: The reference *beam* for the tracking. If no beam is provided, the command looks for a beam attached to the sequence, i.e. the attribute `seq.beam`.[2] (default: `nil`).
Example: `beam = beam 'lhcbeam' { ... }` where ... are the *beam-attributes*.

range
: A *range* specifying the span of the sequence track. If no range is provided, the command looks for a range attached to the sequence, i.e. the attribute `seq.range`. (default: `nil`).
Example: `range = "S.DS.L8.B1/E.DS.R8.B1"`.

dir
: The s-direction of the tracking: `1` forward, `-1` backward. (default: 1).
Example: `dir = -1`.

s0
: A *number* specifying the initial s-position offset. (default: 0 [m]).
Example: `s0 = 5000`.

X0
: A *mappable* (or a list of *mappable*) specifying initial coordinates `{x,px,y,py,t,pt}`, damap, or beta block for each tracked object, i.e. particle or damap. The beta blocks are converted to damaps, while the coordinates are converted to damaps only if `mapdef` is specified, but both will use `mapdef` to setup the damap constructor. Each tracked object may also contain a `beam` to override the reference beam, and a *logical* `nosave` to discard this object from being saved in the mtable. (default: 0).
Example: `X0 = { x=1e-3, px=-1e-5 }`.

O0
: A *mappable* specifying initial coordinates `{x,px,y,py,t,pt}` of the reference orbit around which X0 definitions take place. If it has the attribute `cofind == true`, it will be used as an initial guess to search for the reference closed orbit. (default: 0).
Example: `O0 = { x=1e-4, px=-2e-5, y=-2e-4, py=1e-5 }`.

deltap
: A *number* (or list of *number*) specifying the initial δ_p to convert (using the beam) and add to the `pt` of each tracked particle or damap. (default: `nil`).

[2] Initial coordinates `X0` may override it by providing per particle or damap beam.

Example: `s0 = 5000`.

nturn

A *number* specifying the number of turn to track. (default: 1).

Example: `nturn = 2`.

nstep

A *number* specifying the number of element to track. A negative value will track all elements. (default: -1).

Example: `nstep = 1`.

nslice

A *number* specifying the number of slices or an *iterable* of increasing relative positions or a *callable* `(elm, mflw, lw)` returning one of the two previous kind of positions to track in the elements. The arguments of the callable are in order, the current element, the tracked map flow, and the length weight of the step. This attribute can be locally overridden by the element. (default: 1).

Example: `nslice = 5`.

mapdef

A *logical* or a *damap* specification as defined by the *DAmap* module to track DA maps instead of particles coordinates. A value of `true` is equivalent to invoke the *damap* constructor with `{}` as argument. This attribute allows to track DA maps instead of particles. (default: `nil`).

Example: `mapdef = { xy=2, pt=5 }`.

method

A *number* specifying the order of integration from 1 to 8, or a *string* specifying a special method of integration. Odd orders are rounded to the next even order to select the corresponding Yoshida or Boole integration schemes. The special methods are `simple` (equiv. to `DKD` order 2), `collim` (equiv. to `MKM` order 2), and `teapot` (Teapot splitting order 2). (default: 2).

Example: `method = 'teapot'`.

model

A *string* specifying the integration model, either `'DKD'` for *Drift-Kick-Drift* thin lens integration or `'TKT'` for *Thick-Kick-Thick* thick lens integration.[3] (default: `'TKT'`)

Example: `model = 'DKD'`.

ptcmodel

A *logical* indicating to use strict PTC model.[4] (default: `nil`)

Example: `ptcmodel = true`.

implicit

A *logical* indicating that implicit elements must be sliced too, e.g. for smooth plotting. (default: `false`).

Example: `implicit = true`.

misalign

A *logical* indicating that misalignment must be considered. (default: `false`).

Example: `misalign = true`.

fringe

A *logical* indicating that fringe fields must be considered or a *number* specifying a bit mask to apply

[3] The `TKT` scheme (Yoshida) is automatically converted to the `MKM` scheme (Boole) when approriate.

[4] In all cases, MAD-NG uses PTC setup `time=true, exact=true`.

to all elements fringe flags defined by the element module. The value `true` is equivalent to the bit mask , i.e. allow all elements (default) fringe fields. (default: `true`).

Example: `fringe = false`.

radiate

A *logical* enabling or disabling the radiation or a *string* specifying the type of radiation: `'average'` or `'quantum'`. The value `true` is equivalent to `'average'`. The value `'quantum+photon'` enables the tracking of emitted photons. (default: `false`).

Example: `radiate = 'quantum'`.

totalpath

A *logical* indicating to use the totalpath for the fifth variable `'t'` instead of the local path. (default: `false`).

Example: `totalpath = true`.

save

A *logical* specifying to create a *mtable* and record tracking information at the observation points. The `save` attribute can also be a *string* specifying saving positions in the observed elements: `"atentry"`, `"atslice"`, `"atexit"` (i.e. `true`), `"atbound"` (i.e. entry and exit), `"atbody"` (i.e. slices and exit) and `"atall"`. (default: `true`).

Example: `save = false`.

title

A *string* specifying the title of the *mtable*. If no title is provided, the command looks for the name of the sequence, i.e. the attribute `seq.name`. (default: `nil`).

Example: `title = "track around IP5"`.

observe

A *number* specifying the observation points to consider for recording the tracking information. A zero value will consider all elements, while a positive value will consider selected elements only, checked with method `:is_observed`, every `observe` > 0 turns. (default: `1`).

Example: `observe = 1`.

savesel

A *callable* `(elm, mflw, lw, islc)` acting as a predicate on selected elements for observation, i.e. the element is discarded if the predicate returns `false`. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`)

Example: `savesel = \e -> mylist[e.name] ~= nil`.

savemap

A *logical* indicating to save the damap in the column `__map` of the *mtable*. (default: `false`).

Example: `savemap = true`.

atentry

A *callable* `(elm, mflw, 0, -1)` invoked at element entry. The arguments are in order, the current element, the tracked map flow, zero length and the slice index . (default: `fnil`).

Example: `atentry = myaction`.

atslice

A *callable* `(elm, mflw, lw, islc)` invoked at element slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`).

Example: `atslice = myaction`.

atexit

A *callable* `(elm, mflw, 0, -2)` invoked at element exit. The arguments are in order, the current element, the tracked map flow, zero length and the slice index . (default: `fnil`).

Example: `atexit = myaction`.

ataper

A *callable* `(elm, mflw, lw, islc)` invoked at element aperture checks, by default at last slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. If a particle or a damap hits the aperture, then its `status = "lost"` and it is removed from the list of tracked items. (default: `fnil`).

Example: `ataper = myaction`.

atsave

A *callable* `(elm, mflw, lw, islc)` invoked at element saving steps, by default at exit. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`).

Example: `atsave = myaction`.

atdebug

A *callable* `(elm, mflw, lw, [msg], [...])` invoked at the entry and exit of element maps during the integration steps, i.e. within the slices. The arguments are in order, the current element, the tracked map flow, the length weight of the integration step and a *string* specifying a debugging message, e.g. `"map_name:0"` for entry and `":1"` for exit. If the level `debug` ≥ 4 and `atdebug` is not specified, the default *function* `mdump` is used. In some cases, extra arguments could be passed to the method. (default: `fnil`).

Example: `atdebug = myaction`.

info

A *number* specifying the information level to control the verbosity of the output on the console. (default: `nil`).

Example: `info = 2`.

debug

A *number* specifying the debug level to perform extra assertions and to control the verbosity of the output on the console. (default: `nil`).

Example: `debug = 2`.

usrdef

Any user defined data that will be attached to the tracked map flow, which is internally passed to the elements method `:track` and to their underlying maps. (default: `nil`).

Example: `usrdef = { myvar=somevalue }`.

mflow

An *mflow* containing the current state of a `track` command. If a map flow is provided, all attributes are discarded except `nstep`, `info` and `debug`, as the command was already set up upon its creation. (default: `nil`).

Example: `mflow = mflow0`.

The `track` command returns the following objects in this order:

mtbl

An *mtable* corresponding to the TFS table of the `track` command.

mflw
An *mflow* corresponding to the map flow of the `track` command.

eidx
An optional *number* corresponding to the last tracked element index in the sequence when `nstep` was specified and stopped the command before the end of the `range`.

2 Track mtable

The `track` command returns a *mtable* where the information described hereafter is the default list of fields written to the TFS files.[5]

The header of the *mtable* contains the fields in the default order:

name
The name of the command that created the *mtable*, e.g. `"track"`.

type
The type of the *mtable*, i.e. `"track"`.

title
The value of the command attribute `title`.

origin
The origin of the application that created the *mtable*, e.g. `"MAD 1.0.0 OSX 64"`.

date
The date of the creation of the *mtable*, e.g. `"27/05/20"`.

time
The time of the creation of the *mtable*, e.g. `"19:18:36"`.

refcol
The reference *column* for the *mtable* dictionnary, e.g. `"name"`.

direction
The value of the command attribute `dir`.

observe
The value of the command attribute `observe`.

implicit
The value of the command attribute `implicit`.

misalign
The value of the command attribute `misalign`.

deltap
The value of the command attribute `deltap`.

lost
The number of lost particle(s) or damap(s).

range
The value of the command attribute `range`.[6]

[5] The output of mtable in TFS files can be fully customized by the user.
[6] This field is not saved in the TFS table by default.

__seq
The *sequence* from the command attribute `sequence`.[7] :

The core of the *mtable* contains the columns in the default order:

name
The name of the element.

kind
The kind of the element.

s
The s-position at the end of the element slice.

l
The length from the start of the element to the end of the element slice.

id
The index of the particle or damap as provided in `X0`.

x
The local coordinate x at the s-position.

px
The local coordinate p_x at the s-position.

y
The local coordinate y at the s-position.

py
The local coordinate p_y at the s-position.

t
The local coordinate t at the s-position.

pt
The local coordinate p_t at the s-position.

pc
The reference beam P_0c in which p_t is expressed.

slc
The slice index ranging from `-2` to `nslice`.

turn
The turn number.

tdir
The t-direction of the tracking in the element.

eidx
The index of the element in the sequence.

status
The status of the particle or damap.

__map
The damap at the s-position.Page 106, 7

[7] Fields and columns starting with two underscores are protected data and never saved to TFS files.

3 Dynamical tracking

Fig. 11.1 presents the scheme of the dynamical tracking through an element sliced with `nslice=3`. The actions `atentry` (index `-1`), `atslice` (indexes `0..3`), and `atexit` (index `-2`) are reversed between the forward tracking (`dir=1` with increasing s-position) and the backward tracking (`dir=-1` with decreasing s-position). By default, the action `atsave` is attached to the exit slice and the action `ataper` is attached to the last slice just before exit, i.e. to the last `atslice` action in the tilted frame, and hence they are also both reversed in the backward tracking.

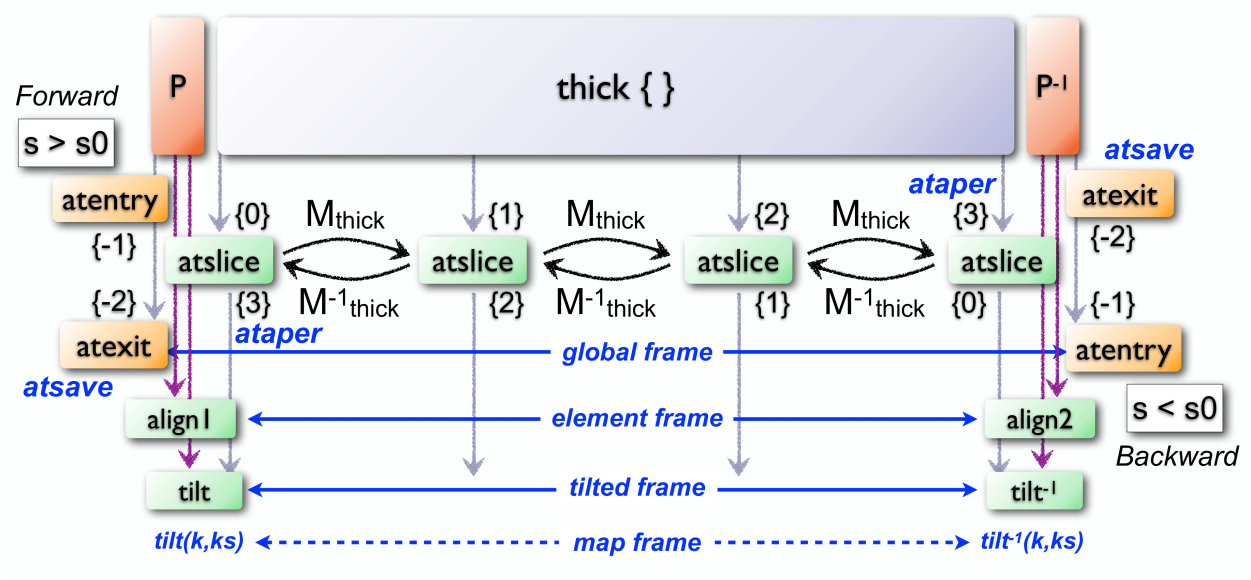

Figure11.1: Dynamical tracking with slices.

3.1 Slicing

The slicing can take three different forms:

- A *number* of the form `nslice=N` that specifies the number of slices with indexes 0..:math:N. This defines a uniform slicing with slice length $l_{\text{slice}} = l_{\text{elem}}/N$.
- An *iterable* of the form `nslice={lw_1,lw_2,..,lw_N}` with $\sum_i lw_i = 1$ that specifies the fraction of length of each slice with indexes 0 .. N where N=`#nslice`. This defines a non-uniform slicing with a slice length of $l_i = lw_i \times l_{\text{elem}}$.
- A *callable* `(elm, mflw, lw)` returning one of the two previous forms of slicing. The arguments are in order, the current element, the tracked map flow, and the length weight of the step, which should allow to return a user-defined element-specific slicing.

The surrounding P and P^{-1} maps represent the patches applied around the body of the element to change the frames, after the `atentry` and before the `atexit` actions:

- The misalignment of the element to move from the *global frame* to the *element frame* if the command attribute `misalign` is set to `true`.
- The tilt of the element to move from the element frame to the *titled frame* if the element attribute `tilt` is non-zero. The `atslice` actions take place in this frame.

The *map frame* is specific to some maps while tracking through the body of the element. In principle, the map frame is not visible to the user, only to the integrator. For example, a quadrupole with both `k1` and `k1s` defined will have a *map frame* tilted by the angle $\alpha = -\frac{1}{2}\tan^{-1}\frac{k1s}{k1}$ attached to its thick map, i.e. the focusing matrix handling only $\tilde{k}_1 = \sqrt{k1^2 + k1s^2}$, but not to its thin map, i.e. the kick from all multipoles (minus `k1` and `k1s`) expressed in the *tilted frame*, during the integration steps.

3.2 Sub-elements

The `track` command takes sub-elements into account. In this case, the slicing specification is taken between sub-elements, e.g. 3 slices with 2 sub-elements gives a final count of 9 slices. It is possible to adjust the number of slices between sub-elements with the third form of slicing specifier, i.e. by using a callable where the length weight argument is between the current (or the end of the element) and the last sub-elements (or the start of the element).

3.3 Particles status

The `track` command initializes the map flow with particles or damaps or both, depending on the attributes `X0` and `mapdef`. The `status` attribute of each particle or damap will be set to one of `"Xset"`, `"Mset"`, and `"Aset"` to track the origin of its initialization: coordinates, damap, or normalizing damap (normal form or beta block). After the tracking, some particles or damaps may have the status `"lost"` and their number being recorded in the counter `lost` from TFS table header. Other commands like `cofind` or `twiss` may add extra tags to the status value, like `"stable"`, `"unstable"` and `"singular"`.

4 Examples

Chapter 12. Cofind

The `cofind` command (i.e. closed orbit finder) provides a simple interface to find a closed orbit using the Newton algorithm on top of the `track` command.

1 Command synopsis

Listing 12.1: Synopsis of the `cofind` command with default setup.

```
mtbl, mflw = cofind} {
        sequence=sequ,  -- sequence (required)
        beam=nil,       -- beam (or sequence.beam, required)
        range=nil,      -- range of tracking (or sequence.range)
        dir=nil,        -- s-direction of tracking (1 or -1)
        s0=nil,         -- initial s-position offset [m]
        X0=nil,         -- initial coordinates (or damap, or beta block)
        O0=nil,         -- initial coordinates of reference orbit
        deltap=nil,     -- initial deltap(s)
        nturn=nil,      -- number of turns to track
        nslice=nil,     -- number of slices (or weights) for each element
        mapdef=true,    -- setup for damap (or list of, true => {})
        method=nil,     -- method or order for integration (1 to 8)
        model=nil,      -- model for integration ('DKD' or 'TKT')
        ptcmodel=nil,   -- use strict PTC thick model (override option)
        implicit=nil,   -- slice implicit elements too (e.g. plots)
        misalign=nil,   -- consider misalignment
        fringe=nil,     -- enable fringe fields (see element.flags.fringe)
        radiate=nil,    -- radiate at slices
        totalpath=nil,  -- variable 't' is the totalpath
        save=false,     -- create mtable and save results
        title=nil,      -- title of mtable (default seq.name)
        observe=nil,    -- save only in observed elements (every n turns)
        savesel=nil,    -- save selector (predicate)
        savemap=nil,    -- save damap in the column __map
        atentry=nil,    -- action called when entering an element
        atslice=nil,    -- action called after each element slices
        atexit=nil,     -- action called when exiting an element
        ataper=nil,     -- action called when checking for aperture
        atsave=nil,     -- action called when saving in mtable
        atdebug=fnil,   -- action called when debugging the element maps
        codiff=1e-10,   -- finite differences step for jacobian
        coiter=20,      -- maximum number of iterations
        cotol=1e-8,     -- closed orbit tolerance (i.e.|dX|)
        X1=0,           -- optional final coordinates translation
        info=nil,       -- information level (output on terminal)
```

(continues on next page)

(continued from previous page)

```
        debug=nil,      -- debug information level (output on terminal)
        usrdef=nil,     -- user defined data attached to the mflow
        mflow=nil,      -- mflow, exclusive with other attributes
}
```

The `cofind` command format is summarized in Listing 12.1, including the default setup of the attributes. Most of these attributes are set to `nil` by default, meaning that `cofind` relies on the `track` command defaults. The `cofind` command supports the following attributes:

sequence

The *sequence* to track. (no default, required).

Example: `sequence = lhcb1`.

beam

The reference *beam* for the tracking. If no beam is provided, the command looks for a beam attached to the sequence, i.e. the attribute `seq.beam`. (default: `nil`)

Example: `beam = beam 'lhcbeam' { beam-attributes }`.[1]

range

A *range* specifying the span of the sequence track. If no range is provided, the command looks for a range attached to the sequence, i.e. the attribute `seq.range`. (default: `nil`).

Example: `range = "S.DS.L8.B1/E.DS.R8.B1"`.

dir

The s-direction of the tracking: `1` forward, `-1` backward. (default: `nil`).

Example: `dir = -1`.

s0

A *number* specifying the initial s-position offset. (default: `nil`).

Example: `s0 = 5000`.

X0

A *mappable* (or a list of *mappable*) specifying initial coordinates `{x,px,y,py, t,pt}`, damap, or beta block for each tracked object, i.e. particle or damap. The beta blocks are converted to damaps, while the coordinates are converted to damaps only if `mapdef` is specified, but both will use `mapdef` to setup the damap constructor. Each tracked object may also contain a `beam` to override the reference beam, and a *logical* `nosave` to discard this object from being saved in the mtable. (default: `nil`).

Example: `X0 = { x=1e-3, px=-1e-5 }`.

O0

A *mappable* specifying initial coordinates `{x,px,y,py,t,pt}` of the reference orbit around which X0 definitions take place. If it has the attribute `cofind == true`, it will be used as an initial guess to search for the reference closed orbit. (default: `0`).

Example: `O0 = { x=1e-4, px=-2e-5, y=-2e-4, py=1e-5 }`.

deltap

A *number* (or list of *number*) specifying the initial δ_p to convert (using the beam) and add to the `pt` of each tracked particle or damap. (default:`nil`).

Example: `s0 = 5000`.

[1] Initial coordinates `X0` may override it by providing a beam per particle or damap.

nturn

A *number* specifying the number of turn to track. (default: `nil`).

Example: `nturn = 2`.

nstep

A *number* specifying the number of element to track. A negative value will track all elements. (default: `nil`).

Example: `nstep = 1`.

nslice

A *number* specifying the number of slices or an *iterable* of increasing relative positions or a *callable* `(elm, mflw, lw)` returning one of the two previous kind of positions to track in the elements. The arguments of the callable are in order, the current element, the tracked map flow, and the length weight of the step. This attribute can be locally overridden by the element. (default: `nil`).

Example: `nslice = 5`.

mapdef

A *logical* or a *damap* specification as defined by the *DAmap* module to track DA maps instead of particles coordinates. A value of `true` is equivalent to invoke the *damap* constructor with `{}` as argument. A value of `false` or `nil` disable the use of damaps and force `cofind` to replace each particles or damaps by seven particles to approximate their Jacobian by finite difference. (default: `true`).

Example: `mapdef = { xy=2, pt=5 }`.

method

A *number* specifying the order of integration from 1 to 8, or a *string* specifying a special method of integration. Odd orders are rounded to the next even order to select the corresponding Yoshida or Boole integration schemes. The special methods are `simple` (equiv. to `DKD` order 2), `collim` (equiv. to `MKM` order 2), and `teapot` (Teapot splitting order 2). (default: `nil`).

Example: `method = 'teapot'`.

model

A *string* specifying the integration model, either `'DKD'` for *Drift-Kick-Drift* thin lens integration or `'TKT'` for *Thick-Kick-Thick* thick lens integration.[2] (default: `nil`)

Example: `model = 'DKD'`.

ptcmodel

A *logical* indicating to use strict PTC model.[3] (default: `nil`)

Example: `ptcmodel = true`.

implicit

A *logical* indicating that implicit elements must be sliced too, e.g. for smooth plotting. (default: `nil`).

Example: `implicit = true`.

misalign

A *logical* indicating that misalignment must be considered. (default: `nil`).

Example: `misalign = true`.

fringe

A *logical* indicating that fringe fields must be considered or a *number* specifying a bit mask to apply

[2] The `TKT` scheme (Yoshida) is automatically converted to the `MKM` scheme (Boole) when appropriate.

[3] In all cases, MAD-NG uses PTC setup `time=true, exact=true`.

to all elements fringe flags defined by the element module. The value `true` is equivalent to the bit mask , i.e. allow all elements (default) fringe fields. (default: `nil`).

Example: `fringe = false`.

radiate

A *logical* enabling or disabling the radiation or the *string* specifying the `'average'` type of radiation. The value `true` is equivalent to `'average'` and the value `'quantum'` is converted to `'average'`. (default: `nil`).

Example: `radiate = 'average'`.

totalpath

A *logical* indicating to use the totalpath for the fifth variable `'t'` instead of the local path. (default: `nil`).

Example: `totalpath = true`.

save

A *logical* specifying to create a *mtable* and record tracking information at the observation points. The `save` attribute can also be a *string* specifying saving positions in the observed elements: `"atentry"`, `"atslice"`, `"atexit"` (i.e. `true`), `"atbound"` (i.e. entry and exit), `"atbody"` (i.e. slices and exit) and `"atall"`. (default: `false`).

Example: `save = false`.

title

A *string* specifying the title of the *mtable*. If no title is provided, the command looks for the name of the sequence, i.e. the attribute `seq.name`. (default: `nil`).

Example: `title = "track around IP5"`.

observe

A *number* specifying the observation points to consider for recording the tracking information. A zero value will consider all elements, while a positive value will consider selected elements only, checked with method `:is_observed`, every `observe`> 0 turns. (default: `nil`).

Example: `observe = 1`.

savesel

A *callable* `(elm, mflw, lw, islc)` acting as a predicate on selected elements for observation, i.e. the element is discarded if the predicate returns `false`. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `nil`)

Example: `savesel = \e -> mylist[e.name] ~= nil`.

savemap

A *logical* indicating to save the damap in the column `__map` of the *mtable*. (default: `nil`).

Example: `savemap = true`.

atentry

A *callable* `(elm, mflw, 0, -1)` invoked at element entry. The arguments are in order, the current element, the tracked map flow, zero length and the slice index `-1`. (default: `nil`).

Example: `atentry = myaction`.

atslice

A *callable* `(elm, mflw, lw, islc)` invoked at element slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `nil`).

Example: `atslice = myaction`.

atexit

A *callable* `(elm, mflw, 0, -2)` invoked at element exit. The arguments are in order, the current element, the tracked map flow, zero length and the slice index . (default: `nil`).

Example: `atexit = myaction`.

ataper

A *callable* `(elm, mflw, lw, islc)` invoked at element aperture checks, by default at last slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. If a particle or a damap hits the aperture, then its `status="lost"` and it is removed from the list of tracked items. (default: `fnil`).

Example: `ataper = myaction`.

atsave

A *callable* `(elm, mflw, lw, islc)` invoked at element saving steps, by default at exit. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `nil`).

Example: `atsave = myaction`.

atdebug

A *callable* `(elm, mflw, lw, [msg], [...])` invoked at the entry and exit of element maps during the integration steps, i.e. within the slices. The arguments are in order, the current element, the tracked map flow, the length weight of the integration step and a *string* specifying a debugging message, e.g. `"map_name:0"` for entry and `":1"` for exit. If the level `debug` ≥ 4 and `atdebug` is not specified, the default *function* `mdump` is used. In some cases, extra arguments could be passed to the method. (default: `fnil`).

Example: `atdebug = myaction`.

codiff

A *number* specifying the finite difference step to approximate the Jacobian when damaps are disabled. If `codiff` is larger than $100\times$`cotol`, it will be adjusted to `cotol` $/100$ and a warning will be emitted. (default: `1e-8`).

Example: `codiff = 1e-10`.

coiter

A *number* specifying the maximum number of iteration. If this threshold is reached, all the remaining tracked objects are tagged as `"unstable"`. (default: 20).

Example: `coiter = 5`.

cotol

A *number* specifying the closed orbit tolerance. If all coordinates update of a particle or a damap are smaller than `cotol`, then it is tagged as `"stable"`. (default: `1e-8`).

Example: `cotol = 1e-6`.

X1

A *mappable* specifying the coordinates `{x,px,y,py,t,pt}` to *subtract* to the final coordinates of the particles or the damaps. (default: `0`).

Example: `X1 = { t=100, pt=10 }`.

info

A *number* specifying the information level to control the verbosity of the output on the console. (default: `nil`).

Example: `info = 2`.

debug
A *number* specifying the debug level to perform extra assertions and to control the verbosity of the output on the console. (default: `nil`).

Example: `debug = 2`.

usrdef
Any user defined data that will be attached to the tracked map flow, which is internally passed to the elements method `:track` and to their underlying maps. (default: `nil`).

Example: `usrdef = { myvar=somevalue }`.

mflow
A *mflow* containing the current state of a `track` command. If a map flow is provided, all attributes are discarded except `nstep`, `info` and `debug`, as the command was already set up upon its creation. (default: `nil`).

Example: `mflow = mflow0`.

The `cofind` command stops when all particles or damap are tagged as `"stable"`, `"unstable"`, `"singular"` or `"lost"`. The `cofind` command returns the following objects in this order:

mtbl
A *mtable* corresponding to the TFS table of the `track` command where the `status` column may also contain the new values `"stable"`, `"unstable"` or `"singular"`.

mflw
A *mflow* corresponding to the map flow of the `track` command. The particles or damaps `status` are tagged and ordered by `"stable"`, `"unstable"`, `"singular"`, `"lost"` and `id`.

2 Cofind mtable

The `cofind` command returns the `track` *mtable* unmodified except for the `status` column. The tracked objects id will appear once per iteration at the `$end` marker, and other defined observation points if any, until they are removed from the list of tracked objects.

3 Examples

TODO

Chapter 13. Twiss

The `twiss` command provides a simple interface to compute the optical functions around an orbit on top of the `track` command, and the `cofind` command if the search for closed orbits is requested.

1 Command synopsis

The `twiss` command format is summarized in Listing 13.1, including the default setup of the attributes. Most of these attributes are set to `nil` by default, meaning that `twiss` relies on the `track` and the `cofind` commands defaults.

Listing 13.1: Synopsis of the `twiss` command with default setup.

```
mtbl, mflw [, eidx] = twiss {
        sequence=sequ,  -- sequence (required)
        beam=nil,       -- beam (or sequence.beam, required)
        range=nil,      -- range of tracking (or sequence.range)
        dir=nil,        -- s-direction of tracking (1 or -1)
        s0=nil,         -- initial s-position offset [m]
        X0=nil,         -- initial coordinates (or damap(s), or beta block(s))
        O0=nil,         -- initial coordinates of reference orbit
        deltap=nil,     -- initial deltap(s)
        chrom=false,    -- compute chromatic functions by finite difference
        coupling=false, -- compute optical functions for non-diagonal modes
        nturn=nil,      -- number of turns to track
        nstep=nil,      -- number of elements to track
        nslice=nil,     -- number of slices (or weights) for each element
        mapdef=true,    -- setup for damap (or list of, true => {})
        method=nil,     -- method or order for integration (1 to 8)
        model=nil,      -- model for integration ('DKD' or 'TKT')
        ptcmodel=nil,   -- use strict PTC thick model (override option)
        implicit=nil,   -- slice implicit elements too (e.g. plots)
        misalign=nil,   -- consider misalignment
        fringe=nil,     -- enable fringe fields (see element.flags.fringe)
        radiate=nil,    -- radiate at slices
        totalpath=nil,  -- variable 't' is the totalpath
        save=true,      -- create mtable and save results
        title=nil,      -- title of mtable (default seq.name)
        observe=0,      -- save only in observed elements (every n turns)
        savesel=nil,    -- save selector (predicate)
        savemap=nil,    -- save damap in the column __map
        atentry=nil,    -- action called when entering an element
        atslice=nil,    -- action called after each element slices
        atexit=nil,     -- action called when exiting an element
        ataper=nil,     -- action called when checking for aperture
        atsave=nil,     -- action called when saving in mtable
```

(continues on next page)

(continued from previous page)

```
        atdebug=fnil,    -- action called when debugging the element maps
        codiff=nil,      -- finite differences step for jacobian
        coiter=nil,      -- maximum number of iterations
        cotol=nil,       -- closed orbit tolerance (i.e.|dX|)
        X1=nil,          -- optional final coordinates translation
        info=nil,        -- information level (output on terminal)
        debug=nil,       -- debug information level (output on terminal)
        usrdef=nil,      -- user defined data attached to the mflow
        mflow=nil,       -- mflow, exclusive with other attributes
}
```

The `twiss` command supports the following attributes:

sequence

The *sequence* to track. (no default, required).

Example: `sequence = lhcb1`.

beam

The reference *beam* for the tracking. If no beam is provided, the command looks for a beam attached to the sequence, i.e. the attribute `seq.beam` .[1] (default: `nil`).

Example: `beam = beam 'lhcbeam' { beam-attributes }`.

range

A *range* specifying the span of the sequence track. If no range is provided, the command looks for a range attached to the sequence, i.e. the attribute `seq.range`. (default: `nil`).

Example: `range = "S.DS.L8.B1/E.DS.R8.B1"`.

dir

The s-direction of the tracking: `1` forward, `-1` backward. (default: `nil`).

Example: `dir = -1`.

s0

A *number* specifying the initial s-position offset. (default: `nil`).

Example: `s0 = 5000`.

X0

A *mappable* (or a list of *mappable*) specifying initial coordinates `{x,px,y,py, t,pt}`, damap, or beta0 block for each tracked object, i.e. particle or damap. The beta0 blocks are converted to damaps, while the coordinates are converted to damaps only if `mapdef` is specified, but both will use `mapdef` to setup the damap constructor. A closed orbit will be automatically searched for damaps built from coordinates. Each tracked object may also contain a `beam` to override the reference beam, and a *logical* `nosave` to discard this object from being saved in the mtable. (default: `0`).

Example: `X0 = { x=1e-3, px=-1e-5 }`.

O0

A *mappable* specifying initial coordinates `{x,px,y,py,t,pt}` of the reference orbit around which X0 definitions take place. If it has the attribute `cofind == true`, it will be used as an initial guess to search for the reference closed orbit. (default: `0`).

Example: `O0 = { x=1e-4, px=-2e-5, y=-2e-4, py=1e-5 }`.

[1] Initial coordinates `X0` may override it by providing a beam per particle or damap.

deltap

A *number* (or list of *number*) specifying the initial δ_p to convert (using the beam) and add to the `pt` of each tracked particle or damap. (default: `nil`).

Example: `s0 = 5000`.

chrom

A *logical* specifying to calculate the chromatic functions by finite different using an extra $\delta_p =$ `1e-6`. (default: `false`).

Example: `chrom = true`.

coupling

A *logical* specifying to calculate the optical functions for coupling terms in the normalized forms. (default: `false`).

Example: `chrom = true`.

nturn

A *number* specifying the number of turn to track. (default: `nil`).

Example: `nturn = 2`.

nstep

A *number* specifying the number of element to track. A negative value will track all elements. (default: `nil`).

Example: `nstep = 1`.

nslice

A *number* specifying the number of slices or an *iterable* of increasing relative positions or a *callable* `(elm, mflw, lw)` returning one of the two previous kind of positions to track in the elements. The arguments of the callable are in order, the current element, the tracked map flow, and the length weight of the step. This attribute can be locally overridden by the element. (default: `nil`).

Example: `nslice = 5`.

mapdef

A *logical* or a *damap* specification as defined by the *DAmap* module to track DA maps instead of particles coordinates. A value of `true` is equivalent to invoke the *damap* constructor with `{}` as argument. A value of `false` or `nil` will be internally forced to `true` for the tracking of the normalized forms. (default: `true`).

Example: `mapdef = { xy=2, pt=5 }`.

method

A *number* specifying the order of integration from 1 to 8, or a *string* specifying a special method of integration. Odd orders are rounded to the next even order to select the corresponding Yoshida or Boole integration schemes. The special methods are `simple` (equiv. to `DKD` order 2), `collim` (equiv. to `MKM` order 2), and `teapot` (Teapot splitting order 2). (default: `nil`).

Example: `method = 'teapot'`.

model

A *string* specifying the integration model, either `'DKD'` for *Drift-Kick-Drift* thin lens integration or `'TKT'` for *Thick-Kick-Thick* thick lens integration.[2] (default: `nil`)

Example: `model = 'DKD'`.

ptcmodel

[2] The `TKT` scheme (Yoshida) is automatically converted to the `MKM` scheme (Boole) when appropriate.

A *logical* indicating to use strict PTC model.[3] (default: `nil`)

Example: `ptcmodel = true`.

implicit

A *logical* indicating that implicit elements must be sliced too, e.g. for smooth plotting. (default: `nil`).

Example: `implicit = true`.

misalign

A *logical* indicating that misalignment must be considered. (default: `nil`).

Example: `misalign = true`.

fringe

A *logical* indicating that fringe fields must be considered or a *number* specifying a bit mask to apply to all elements fringe flags defined by the element module. The value `true` is equivalent to the bit mask , i.e. allow all elements (default) fringe fields. (default: `nil`).

Example: `fringe = false`.

radiate

A *logical* enabling or disabling the radiation or the *string* specifying the `'average'` type of radiation during the closed orbit search. The value `true` is equivalent to `'average'` and the value `'quantum'` is converted to `'average'`. (default: `nil`).

Example: `radiate = 'average'`.

totalpath

A *logical* indicating to use the totalpath for the fifth variable `'t'` instead of the local path. (default: `nil`).

Example: `totalpath = true`.

save

A *logical* specifying to create a *mtable* and record tracking information at the observation points. The `save` attribute can also be a *string* specifying saving positions in the observed elements: `"atentry"`, `"atslice"`, `"atexit"` (i.e. `true`), `"atbound"` (i.e. entry and exit), `"atbody"` (i.e. slices and exit) and `"atall"`. (default: `false`).

Example: `save = false`.

title

A *string* specifying the title of the *mtable*. If no title is provided, the command looks for the name of the sequence, i.e. the attribute `seq.name`. (default: `nil`).

Example: `title = "track around IP5"`.

observe

A *number* specifying the observation points to consider for recording the tracking information. A zero value will consider all elements, while a positive value will consider selected elements only, checked with method `:is_observed`, every `observe`> 0 turns. (default: `nil`).

Example: `observe = 1`.

savesel

A *callable* `(elm, mflw, lw, islc)` acting as a predicate on selected elements for observation, i.e. the element is discarded if the predicate returns `false`. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`)

Example: `savesel = \e -> mylist[e.name] ~= nil`.

[3] In all cases, MAD-NG uses PTC setup `time=true, exact=true`.

savemap

A *logical* indicating to save the damap in the column `__map` of the *mtable*. (default: `nil`).

Example: `savemap = true`.

atentry

A *callable* `(elm, mflw, 0, -1)` invoked at element entry. The arguments are in order, the current element, the tracked map flow, zero length and the slice index `-1`. (default: `fnil`).

Example: `atentry = myaction`.

atslice

A *callable* `(elm, mflw, lw, islc)` invoked at element slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`).

Example: `atslice = myaction`.

atexit

A *callable* `(elm, mflw, 0, -2)` invoked at element exit. The arguments are in order, the current element, the tracked map flow, zero length and the slice index . (default: `fnil`).

Example: `atexit = myaction`.

ataper

A *callable* `(elm, mflw, lw, islc)` invoked at element aperture checks, by default at last slice. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. If a particle or a damap hits the aperture, then its `status="lost"` and it is removed from the list of tracked items. (default: `fnil`).

Example: `ataper = myaction`.

atsave

A *callable* `(elm, mflw, lw, islc)` invoked at element saving steps, by default at exit. The arguments are in order, the current element, the tracked map flow, the length weight of the slice and the slice index. (default: `fnil`).

Example: `atsave = myaction`.

atdebug

A *callable* `(elm, mflw, lw, [msg], [...])` invoked at the entry and exit of element maps during the integration steps, i.e. within the slices. The arguments are in order, the current element, the tracked map flow, the length weight of the integration step and a *string* specifying a debugging message, e.g. `"map_name:0"` for entry and `":1"` for exit. If the level `debug` ≥ 4 and `atdebug` is not specified, the default *function* `mdump` is used. In some cases, extra arguments could be passed to the method. (default: `fnil`).

Example: `atdebug = myaction`.

codiff

A *number* specifying the finite difference step to approximate the Jacobian when damaps are disabled. If `codiff` is larger than $100\times$`cotol`, it will be adjusted to `cotol` $/100$ and a warning will be emitted. (default: `1e-8`).

Example: `codiff = 1e-10`.

coiter

A *number* specifying the maximum number of iteration. If this threshold is reached, all the remaining tracked objects are tagged as `"unstable"`. (default: 20).

Example: `coiter = 5`.

cotol

A *number* specifying the closed orbit tolerance. If all coordinates update of a particle or a damap are smaller than `cotol`, then it is tagged as `"stable"`. (default: `1e-8`).

Example: `cotol = 1e-6`.

X1

A *mappable* specifying the coordinates `{x,px,y,py,t,pt}` to *subtract* to the final coordinates of the particles or the damaps. (default: `0`).

Example: `X1 = { t=100, pt=10 }`.

info

A *number* specifying the information level to control the verbosity of the output on the console. (default: `nil`).

Example: `info = 2`.

debug

A *number* specifying the debug level to perform extra assertions and to control the verbosity of the output on the console. (default: `nil`).

Example: `debug = 2`.

usrdef

Any user defined data that will be attached to the tracked map flow, which is internally passed to the elements method `:track` and to their underlying maps. (default: `nil`).

Example: `usrdef = { myvar=somevalue }`.

mflow

A *mflow* containing the current state of a `track` command. If a map flow is provided, all attributes are discarded except `nstep`, `info` and `debug`, as the command was already set up upon its creation. (default: `nil`).

Example: `mflow = mflow0`.

The `twiss` command returns the following objects in this order:

mtbl

A *mtable* corresponding to the augmented TFS table of the `track` command with the `twiss` command columns.

mflw

A *mflow* corresponding to the augmented map flow of the `track` command with the `twiss` command data.

eidx

An optional *number* corresponding to the last tracked element index in the sequence when `nstep` was specified and stopped the command before the end of the `range`.

2 Twiss mtable

The `twiss` command returns a *mtable* where the information described hereafter is the default list of fields written to the TFS files.[4]

The header of the *mtable* contains the fields in the default order:[5]

name
The name of the command that created the *mtable*, e.g. `"track"`.

type
The type of the *mtable*, i.e. `"track"`.

title
The value of the command attribute `title`.

origin
The origin of the application that created the *mtable*, e.g. `"MAD 1.0.0 OSX 64"`.

date
The date of the creation of the *mtable*, e.g. `"27/05/20"`.

time
The time of the creation of the *mtable*, e.g. `"19:18:36"`.

refcol
The reference *column* for the *mtable* dictionnary, e.g. `"name"`.

direction
The value of the command attribute `dir`.

observe
The value of the command attribute `observe`.

implicit
The value of the command attribute `implicit`.

misalign
The value of the command attribute `misalign`.

deltap
The value of the command attribute `deltap`.

lost
The number of lost particle(s) or damap(s).

chrom
The value of the command attribute `chrom`.

coupling
The value of the command attribute `coupling`.

length
The s-length of the tracked design orbit.

q1
The tunes of mode 1.

[4] The output of mtable in TFS files can be fully customized by the user.
[5] The fields from `name` to `lost` are set by the `track` command

q2
The tunes of mode 2.

q3
The tunes of mode 3.

alfap
The momentum compaction factor α_p.

etap
The phase slip factor η_p.

gammatr
The energy gamma transition γ_{tr}.

synch_1
The first synchroton radiation integral.

synch_2
The second synchroton radiation integral.

synch_3
The third synchroton radiation integral.

synch_4
The fourth synchroton radiation integral.

synch_5
The fifth synchroton radiation integral.

synch_6
The sixth synchroton radiation integral.

synch_8
The eighth synchroton radiation integral.

range
The value of the command attribute `range`.[6]

__seq
The *sequence* from the command attribute `sequence`.[7]

The core of the *mtable* contains the columns in the default order:[8]

name
The name of the element.

kind
The kind of the element.

s
The s-position at the end of the element slice.

l
The length from the start of the element to the end of the element slice.

id
The index of the particle or damap as provided in `X0`.

[6] This field is not saved in the TFS table by default.
[7] Fields and columns starting with two underscores are protected data and never saved to TFS files.
[8] The column from `name` to `status` are set by the `track` command.

x
The local coordinate x at the s-position .

px
The local coordinate p_x at the s-position.

y
The local coordinate y at the s-position.

py
The local coordinate p_y at the s-position.

t
The local coordinate t at the s-position.

pt
The local coordinate p_t at the s-position.

slc
The slice index ranging from `-2` to `nslice`.

turn
The turn number.

tdir
The t-direction of the tracking in the element.

eidx
The index of the element in the sequence.

status
The status of the particle or damap.

alfa11
The optical function α of mode 1 at the s-position.

beta11
The optical function β of mode 1 at the s-position.

gama11
The optical function γ of mode 1 at the s-position.

mu1
The phase advance μ of mode 1 at the s-position.

dx
The dispersion function of x at the s-position.

dpx
The dispersion function of p_x at the s-position.

alfa22
The optical function α of mode 2 at the s-position.

beta22
The optical function β of mode 2 at the s-position.

gama22
The optical function γ of mode 2 at the s-position.

mu2
The phase advance μ of mode 2 at the s-position.

dy
The dispersion function of y at the s-position.

dpy
The dispersion function of p_y at the s-position.

alfa33
The optical function α of mode 3 at the s-position.

beta33
The optical function β of mode 3 at the s-position.

gama33
The optical function γ of mode 3 at the s-position.

mu3
The phase advance μ of mode 3 at the s-position.

__map
The damap at the s-position.Page 122, 7

The `chrom` attribute will add the following fields to the *mtable* header:

dq1
The chromatic derivative of tunes of mode 1, i.e. chromaticities.

dq2
The chromatic derivative of tunes of mode 2, i.e. chromaticities.

dq3
The chromatic derivative of tunes of mode 3, i.e. chromaticities.

The `chrom` attribute will add the following columns to the *mtable*:

dmu1
The chromatic derivative of the phase advance of mode 1 at the s-position.

ddx
The chromatic derivative of the dispersion function of x at the s-position.

ddpx
The chromatic derivative of the dispersion function of p_x at the s-position.

wx
The chromatic amplitude function of mode 1 at the s-position.

phix
The chromatic phase function of mode 1 at the s-position.

dmu2
The chromatic derivative of the phase advance of mode 2 at the s-position.

ddy
The chromatic derivative of the dispersion function of y at the s-position.

ddpy
The chromatic derivative of the dispersion function of p_y at the s-position.

wy
The chromatic amplitude function of mode 2 at the s-position.

phiy

The chromatic phase function of mode 2 at the s-position.

The `coupling` attribute will add the following columns to the *mtable*:

alfa12
The optical function α of coupling mode 1-2 at the s-position.

beta12
The optical function β of coupling mode 1-2 at the s-position.

gama12
The optical function γ of coupling mode 1-2 at the s-position.

alfa13
The optical function α of coupling mode 1-3 at the s-position.

beta13
The optical function β of coupling mode 1-3 at the s-position.

gama13
The optical function γ of coupling mode 1-3 at the s-position.

alfa21
The optical function α of coupling mode 2-1 at the s-position.

beta21
The optical function β of coupling mode 2-1 at the s-position.

gama21
The optical function γ of coupling mode 2-1 at the s-position.

alfa23
The optical function α of coupling mode 2-3 at the s-position.

beta23
The optical function β of coupling mode 2-3 at the s-position.

gama23
The optical function γ of coupling mode 2-3 at the s-position.

alfa31
The optical function α of coupling mode 3-1 at the s-position.

beta31
The optical function β of coupling mode 3-1 at the s-position.

gama31
The optical function γ of coupling mode 3-1 at the s-position.

alfa32
The optical function α of coupling mode 3-2 at the s-position.

beta32
The optical function β of coupling mode 3-2 at the s-position.

gama32
The optical function γ of coupling mode 3-2 at the s-position.

3 Tracking linear normal form

TODO

4 Examples

TODO

Chapter 14. Match

The match command provides a unified interface to several optimizer. It can be used to match optics parameters (its main purpose), to fit data sets with parametric functions in the least-squares sense, or to find local or global minima of non-linear problems. Most local methods support bounds, equalities and inequalities constraints. The *least-squares* methods are custom variant of the Newton-Raphson and the Gauss-Newton algorithms implemented by the *LSopt* module. The local and global *non-linear* methods are relying on the *NLopt* module, which interfaces the embedded NLopt library that implements a dozen of well-known algorithms.

1 Command synopsis

The match command format is summarized in Listing 14.1. including the default setup of the attributes.

Listing 14.1: Synopsis of the match command with default setup.

```
status, fmin, ncall = match {
        command         = function or nil,
        variables       = { variables-attributes,
                            { variable-attributes },
                            ..., more variable definitions, ...
                            { variable-attributes } },
        equalities      = { constraints-attributes,
                            { constraint-attributes },
                            ..., more equality definitions, ...
                            { constraint-attributes } },
        inequalities    = { constraints-attributes,
                            { constraint-attributes },
                            ..., more inequality definitions,...
                            { constraint-attributes } },
        weights         = { weights-list },
        objective       = {  objective-attributes },
        maxcall=nil,    -- call limit
        maxtime=nil,    -- time limit
        info=nil,       -- information level (output on terminal)
        debug=nil,      -- debug information level (output on terminal)
        usrdef=nil,     -- user defined data attached to the environment
}
```

The match command supports the following attributes:

command

A *callable* (e) that will be invoked during the optimization process at each iteration. (default: nil).

Example: command := twiss { twiss-attributes }.

variables

An *mappable* of single *variable* specification that can be combined with a *set* of specifications for all variables. (no default, required).

Example: `variables = {{ var="seq.knobs.mq_k1" }}`.

equalities

An *mappable* of single equality specification that can be combined with a *set* of specifications for all equalities. (default: `{}`).

Example: `equalities = {{ expr=\t -> t.q1-64.295, name='q1' }}`.

inequalities

An *mappable* of single inequality specification that can be combined with a *set* of specifications for all inequalities. (default: `{}`).

Example: `inequalities = {{ expr=\t -> t.mq4.beta11-50 }}`.

weights

A *mappable* of weights specification that can be used in the `kind` attribute of the constraints specifications. (default: `{}`).

Example: `weights = { px=10 }`.

objective

A *mappable* of specifications for the objective to minimize. (default: `{}`).

Example: `objective = { method="LD_LMDIF", fmin=1e-10 }`.

maxcall

A *number* specifying the maximum allowed calls of the `command` function or the `objective` function. (default: `nil`).

Example: `maxcall = 100`.

maxtime

A *number* specifying the maximum allowed time in seconds. (default: `nil`).

Example: `maxtime = 60`.

info

A *number* specifying the information level to control the verbosity of the output on the *console*. (default: `nil`).

Example: `info = 3`.

debug

A *number* specifying the debug level to perform extra assertions and to control the verbosity of the output on the *console*. (default: `nil`).

Example: `debug = 2`.

usrdef

Any user defined data that will be attached to the matching environment, which is passed as extra argument to all user defined functions in the `match` command. (default: `nil`).

Example: `usrdef = { var=vector(15) }`.

The `match` command returns the following values in this order:

status

A *string* corresponding to the status of the command or the stopping reason of the method. See Table 14.1 for the list of supported status.

fmin

A *number* corresponding to the best minimum reached during the optimization.

ncall

The *number* of calls of the `command` function or the `objective` function.

Table14.1: List of `status` (*string*) returned by the `match` command.

Status	Meaning
SUCCESS	Generic success (*NLopt* only, unlikely).
FMIN	`fmin` *criteria* is fulfilled by the objective function.
FTOL	`tol` or `rtol` *criteria* are fulfilled by the objective function.
XTOL	`tol` or `rtol` *criteria* are fulfilled by the variables step.
MAXCALL	`maxcall` *criteria* is reached.
MAXTIME	`maxtime` *criteria* is reached.
ROUNDOFF	Round off limited iteration progress, results may still be useful.
STOPPED	Termination forced by user, i.e. `{env.stop = true}`.
	Errors
FAILURE	Generic failure (*NLopt* only, unlikely).
INVALID_ARGS	Invalid argument (*NLopt* only, unlikely).
OUT_OF_MEMORY	Ran out of memory (*NLopt* only, unlikely).

2 Environment

The `match` command creates a matching environment, which is passed as argument to user's functions invoked during an iteration. It contains some useful attributes that can be read or changed during the optimization process (with care):

ncall
: The current *number* of calls of the `command` and/or the `objective` functions.

dtime
: A *number* reporting the current elapsed time.

stop
: A *logical* stopping the `match` command immediately if set to `true`.

info
: The current information level ≥ 0.

debug
: The current debugging level ≥ 0.

usrdef
: The `usrdef` attribute of the `match` command or `nil`.

command
: The `command` attribute of the `match` command or `nil`.

variables
: The `variables` attribute of the `match` command.

equalities
: The `equalities` attribute of the `match` command or `{}`.

inequalities
: The `inequalities` attribute of the `match` command or `{}`.

weights
The `weights` attribute of the `match` command or `{}`.

3 Command

The attribute `command` (default: `nil`) must be a *callable* `(e)` that will be invoked with the matching environment as first argument during the optimization, right after the update of the *variables* to their new values, and before the evaluation of the *constraints* and the *objective* function. (default: `nil`).

```
command = function or nil
```

The value returned by `command` is passed as the first argument to all constraints. If this return value is `nil`, the `match` command considers the current iteration as invalid. Depending on the selected method, the optimizer can start a new iteration or stop.

A typical `command` definition for matching optics is a function that calls a `twiss` command[1] :

```
command := mchklost( twiss { twiss-attributes } )
```

where the function `mchklost` surrounding the `twiss` command checks if the returned `mtable` (i.e. the twiss table) has lost particles and returns `nil`instead:

```
mchklost = \mt -> mt.lost == 0 and mt or nil
```

The function `mchklost`[2] is useful to avoid that all constraints do the check individually.

4 Variables

The attribute `variables` (no default, required) defines the variables that the command `match` will update while trying to minimize the objective function.

```
variables = { variables-attributes,
  { variable-attributes  },
  ... ,more variable definitions, ...
  { variable-attributes  } }
```

The *variable-attributes* is a set of attributes that specify a single variable:

var
A *string* specifying the identifier (and indirection) needed to reach the variable from the user's scope where the `match` command is defined. (default: `nil`).
Example: `var = "lhcb1.mq_12l4_b1.k1"`.

name
A *string* specifying the name of the variable to display when the `info` level is positive. (default: `var`).
Example: `name = "MQ.12L4.B1->k1"`.

[1] Here, the function (i.e. the deferred expression) ignores the matching environment passed as first argument.
[2] The function `mchklost` is provided by the *GPhys module.*

min

A *number* specifying the lower bound for the variable. (default: `-inf`).

Example: `min = -4`.

max

A *number* specifying the upper bound for the variable. (default: `+inf`).

Example: `max = 10`.

sign

A *logical* enforcing the sign of the variable by moving `min` or `max` to zero depending on the sign of its initial value. (default: `false`).

Example: `sign = true`.

slope

A *number* enforcing (*LSopt* methods only) with its sign the variation direction of the variable, i.e. positive will only increase and negative will only decrease. (default: `0`).

Example: `slope = -1`.

step

A small positive *number* used to approximate the derivatives using the *Derivatives* method. If the value is not provided, the command will use some heuristic. (default: `nil`).

Example: `step = 1e-6`.

tol

A *number* specifying the tolerance on the variable step. If an update is smaller than `tol`, the command will return the status `"XTOL"`. (default: `0`).

Example: `tol = 1e-8`.

get

A *callable* `(e)` returning the variable value as a *number*, optionally using the matching environment passed as first argument. This attribute is required if the variable is *local* or an *upvalue* to avoid a significant slowdown of the code. (default: `nil`).

Example: `get := lhcb1.mq_12l4_b1.k1`.

set

A *callable* `(v, e)` updating the variable value with the *number* passed as first argument, optionally using the matching environment passed as second argument.This attribute is required if the variable is *local* or an *upvalue* to avoid a significant slowdown of the code. (default: `nil`).

Example: `set = \v,e => lhcb1.mqxa_1l5.k1 = v*e.usrdef.xon end`.

The *variables-attributes* is a set of attributes that specify all variables together, but with a lower precedence than the single variable specification of the same name unless otherwise specified:

min

Idem *variable-attributes*, but for all variables with no local override.

max

Idem *variable-attributes*, but for all variables with no local override.

sign

Idem *variable-attributes*, but for all variables with no local override.

slope

Idem *variable-attributes*, but for all variables with no local override.

step
Idem *variable-attributes*, but for all variables with no local override.

tol
Idem *variable-attributes*, but for all variables with no local override.

rtol
A *number* specifying the relative tolerance on all variable steps. If an update is smaller than `rtol` relative to its variable value, the command will return the status `"XTOL"`. (default: `eps`).
Example: `tol = 1e-8`.

nvar
A *number* specifying the number of variables of the problem. It is useful when the problem is made abstract with functions and it is not possible to deduce this count from single variable definitions, or one needs to override it. (default: `nil`).
Example: `nvar = 15`.

get
A *callable* `(x, e)` updating a *vector* passed as first argument with the values of all variables, optionally using the matching environment passed as second argument. This attribute supersedes all single variable `get` and may be useful when it is better to read all the variables together, or when they are all *local*s or *upvalue*s. (default: `nil`).
Example: `get = \x,e -> e.usrdef.var:copy(x)`.

set
A *callable* `(x, e)` updating all the variables with the values passed as first argument in a *vector*, optionally using the matching environment passed as second argument. This attribute supersedes all single variable `set` and may be useful when it is better to update all the variables together, or when they are all *local*s or *upvalue*s.(default: `nil`).
Example: `set = \x,e -> x:copy(e.usrdef.var)`.

nowarn
A *logical* disabling a warning emitted when the definition of `get` and `set` are advised but not defined. It is safe to not define `get` and `set` in such case, but it will significantly slowdown the code. (default: `nil`).
Example: `nowarn = true`.

5 Constraints

The attributes `equalities` (default: `{}`) and `inequalities` (default: `{}`) define the constraints that the command `match` will try to satisfy while minimizing the objective function. Equalities and inequalities are considered differently when calculating the *penalty function*.

```
equalities = { constraints-attributes,
               { constraint-attributes } ,
               ... more equality definitions ...
               { constraint-attributes } },

inequalities = { constraints-attributes,
```

(continues on next page)

(continued from previous page)

```
                { constraint-attributes } ,
                ... more inequality definitions ...
                { constraint-attributes } },

weights = { weights-list },
```

The *constraint-attributes* is a set of attributes that specify a single constraint, either an *equality* or an *inequality*:

expr
: A *callable* `(r, e)` returning the constraint value as a *number*, optionally using the result of `command` passed as first argument, and the matching environment passed as second argument. (default: `nil`)

  Example: `expr = \t -> t.IP8.beta11-beta_ip8`.

name
: A *string* specifying the name of the constraint to display when the `info` level is positive. (default: `nil`).

  Example: `name = "betx@IP8"`.

kind
: A *string* specifying the kind to refer to for the weight of the constraint, taken either in the user-defined or in the default *weights-list*. (default: `nil`).

  Example: `kind = "dq1"`.

weight
: A *number* used to override the weight of the constraint. (default: `nil`).

  Example: `weight = 100`.

tol
: A *number* specifying the tolerance to apply on the constraint when checking for its fulfillment. (default: `1e-8`).

  Example: `tol = 1e-6`.

The *constraints-attributes* is a set of attributes that specify all equalities or inequalities constraints together, but with a lower precedence than the single constraint specification of the same name unless otherwise specified:

tol
: Idem *constraint-attributes*, but for all constraints with no local override.

nequ
: A *number* specifying the number of equations (i.e. number of equalities or inequalities) of the problem. It is useful when the problem is made abstract with functions and it is not possible to deduce this count from single constraint definitions, or one needs to override it. (default: `nil`).

  Example: `nequ = 15`.

exec
: A *callable* `(x, c, cjac)` updating a *vector* passed as second argument with the values of all constraints, and updating an optional *matrix* passed as third argument with the Jacobian of all constraints (if not `nil`), using the variables values passed in a *vector* as first argument.

This attribute supersedes all constraints `expr` and may be useful when it is better to update all the constraints together. (default: `nil`).

Example: `exec = myinequ`, where (`nvar=2` and `nequ=2`)

```
local function myinequ (x, c, cjac)
        c:fill { 8*x[1]^3 - x[2] ; (1 - x[1])^3 - x[2] }
        if cjac then -- fill [2x2] matrix if present
                cjac:fill { 24*x[1]^2, - 1 ; - 3*(1 - x[1])^2, - 1 }
        end
end
```

disp

A *logical* disabling the display of the equalities in the summary if it is explicitly set to `false`. This is useful for fitting data where equalities are used to compute the residuals. (default: `nil`).

Example: `disp = false`.

The *weights-list* is a set of attributes that specify weights for kinds used by constraints. It allows to override the default weights of the supported kinds summarized in Table 14.2, or to extend this list with new kinds and weights. The default weight for any undefined `kind` is 1. Example: `weights = { q1=100, q2=100, mykind=3 }`.

Table14.2: List of supported kinds (*string*) and their default weights (*number*).

Name	Weight	Name	Weight	Name	Weight	Generic name
x	10	y	10	t	10	
px	100	py	100	pt	100	
dx	10	dy	10	dt	10	d
dpx	100	dpy	100	dpt	100	dp
ddx	10	ddy	10	ddt	10	dd
ddpx	100	ddpy	100	ddpt	100	ddp
wx	1	wy	1	wz	1	w
phix	1	phiy	1	phiz	1	phi
betx	1	bety	1	betz	1	beta
alfx	10	alfy	10	alfz	10	alfa
mux	10	muy	10	muz	10	mu
beta1	1	beta2	1	beta3	1	beta
alfa1	10	alfa2	10	alfa3	10	alfa
mu1	10	mu2	10	mu3	10	mu
q1	10	q2	10	q3	10	q
dq1	1	dq2	1	dq3	1	dq

6 Objective

The attribute `objective` (default: `{}`) defines the objective that the command `match` will try to minimize.

```
objective = { objective-attributes },
```

The *objective-attributes* is a set of attributes that specify the objective to fulfill:

method

A *string* specifying the algorithm to use for solving the problem, see Table 14.3, Table 14.4 and Table 14.5. (default: `"LN_COBYLA"` if `objective.exec` is defined, `"LD_JACOBIAN"` otherwise).

Example: `method = "LD_LMDIF"`.

submethod

A *string* specifying the algorithm from NLopt module to use for solving the problem locally when the method is an augmented algorithm, see Table 14.4 and Table 14.5 (default: `"LN_COBYLA"`).

Example: `method = "AUGLAG", submethod = "LD_SLSQP"`.

fmin

A *number* corresponding to the minimum to reach during the optimization. For least squares problems, it corresponds to the tolerance on the *penalty function*. If an iteration finds a value smaller than `fmin` and all the constraints are fulfilled, the command will return the status `"FMIN"` . (default: `nil`).

Example: `fmin = 1e-12`.

tol

A *number* specifying the tolerance on the objective function step. If an update is smaller than `tol`, the command will return the status `"FTOL"`. (default: `0`).

Example: `tol = 1e-10`.

rtol

A *number* specifying the relative tolerance on the objective function step. If an update is smaller than `rtol` relative to its step value, the command will return the status `"FTOL"` (default: `0`).

Example: `tol = 1e-8`.

bstra

A *number* specifying the strategy to select the *best case* of the *objective* function. (default: `nil`).

Example: `bstra = 0`.[3]

broyden

A *logical* allowing the Jacobian approximation by finite difference to update its columns with a *Broyden's rank one* estimates when the step of the corresponding variable is almost collinear with the variables step vector. This option may save some expensive calls to `command`, e.g. save Twiss calculations, when it does not degrade the rate of convergence of the selected method. (default: `nil`).

Example: `broyden = true`.

[3] MAD-X matching corresponds to `bstra=0`.

reset

A *logical* specifying to the `match` command to restore the initial state of the variables before returning. This is useful to attempt an optimization without changing the state of the variables. Note that if any function amongst `command`, variables `get` and `set`, constraints `expr` or `exec`, or objective `exec` have side effects on the environment, these will be persistent. (default: `nil`).

Example: `reset = true`.

exec

A *callable* `(x, fgrd)` returning the value of the objective function as a *number*, and updating a *vector* passed as second argument with its gradient, using the variables values passed in a *vector* as first argument. (default: `nil`).

Example: `exec = myfun`, where (`nvar=2`)

```
local function myfun(x, fgrd)
        if fgrd then -- fill [2x1] vector if present
                fgrd:fill { 0, 0.5/sqrt(x[2]) }
        end
        return sqrt(x[2])

end
```

grad

A *logical* enabling (`true`) or disabling (`false`) the approximation by finite difference of the gradient of the objective function or the Jacobian of the constraints. A `nil` value will be converted to `true` if no `exec` function is defined and the selected `method` requires derivatives (`D`), otherwise it will be converted to `false`. (default: `nil`).

Example: `grad = false`.

bisec

A *number* specifying (*LSopt* methods only) the maximum number of attempt to minimize an increasing objective function by reducing the variables steps by half, i.e. that is a *line search* using $\alpha = 0.5^k$ where $k = 0..$bisec. (default: `3` if `objective.exec` is undefined, `0` otherwise).

Example: `bisec = 9`.

rcond

A *number* specifying (*LSopt* methods only) how to determine the effective rank of the Jacobian while solving the least squares system (see `ssolve` from the *Linear Algebra* module). This attribute can be updated between iterations, e.g. through `env.objective.rcond`. (default: `eps`).

Example: `rcond = 1e-14`.

jtol

A *number* specifying (*LSopt* methods only) the tolerance on the norm of the Jacobian rows to reject useless constraints. This attribute can be updated between iterations, e.g. through `env.objective.jtol`. (default: `eps`).

Example: `tol = 1e-14`.

jiter

A *number* specifying (*LSopt* methods only) the maximum allowed attempts to solve the least squares system when variables are rejected, e.g. wrong slope or out-of-bound values. (default: `10`).

Example: `jiter = 15`.

jstra

A *number* specifying (*LSopt* methods only) the strategy to use for reducing the variables of the least squares system. (default: `1`).

Example: `jstra = 3`.[4]

jstra	Strategy for reducing variables of least squares system.
0	no variables reduction, constraints reduction is still active.
1	reduce system variables for bad slopes and out-of-bound values.
2	idem 1, but bad slopes reinitialize variables to their original state.
3	idem 2, but strategy switches definitely to 0 if `jiter` is reached.

7 Algorithms

The `match` command supports local and global optimization algorithms through the `method` attribute, as well as combinations of them with the `submethod` attribute (see *objective*). The method should be selected according to the kind of problem that will add a prefix to the method name: local (`L`) or global (`G`), with (`D`) or without (`N`) derivatives, and least squares or nonlinear function minimization. When the method requires the derivatives (`D`) and no `objective.exec` function is defined or the attribute `grad` is set to `false`, the `match` command will approximate the derivatives, i.e. gradient and Jacobian, by the finite difference method (see *derivatives*).

Most global optimization algorithms explore the variables domain with methods belonging to stochastic sampling, deterministic scanning, and splitting strategies, or a mix of them. Hence, all global methods require *boundaries* to define the searching region, which may or may not be internally scaled to a hypercube. Some global methods allow to specify with the `submethod` attribute, the local method to use for searching local minima. If this is not the case, it is wise to refine the global solution with a local method afterward, as global methods put more effort on finding global solutions than precise local minima. The global (`G`) optimization algorithms, with (`D`) or without (`N`) derivatives, are listed in Table 14.5.

Most local optimization algorithms with derivatives are variants of the Newton iterative method suitable for finding local minima of nonlinear vector-valued function $\vec{f}(\vec{x})$, i.e. searching for stationary points. The iteration steps $\vec{h}$ are given by the minimization $\vec{h} = -\alpha(\nabla^2\vec{f})^{-1}\nabla\vec{f}$, coming from the local approximation of the function at the point $\vec{x}+\vec{h}$ by its Taylor series truncated at second order $\vec{f}(\vec{x}+\vec{h}) \approx \vec{f}(\vec{x}) + \vec{h}^T\nabla\vec{f}(\vec{x}) + \frac{1}{2}\vec{h}^T\nabla^2\vec{f}(\vec{x})\vec{h}$, and solved for $\nabla_{\vec{h}}\vec{f} = 0$. The factor $\alpha > 0$ is part of the line search strategy , which is sometimes replaced or combined with a trusted region strategy like in the Leverberg-Marquardt algorithm. The local (`L`) optimization algorithms, with (`D`) or without (`N`) derivatives, are listed in Table 14.3 for least squares methods and in Table 14.4 for non-linear methods, and can be grouped by family of algorithms:

Newton

An iterative method to solve nonlinear systems that uses iteration step given by the minimization $\vec{h} =$

[4] MAD-X `JACOBIAN` with `strategy=3` corresponds to `jstra=3`.

$-\alpha(\nabla^2\vec{f})^{-1}\nabla\vec{f}$.

Newton-Raphson

An iterative method to solve nonlinear systems that uses iteration step given by the minimization $\vec{h} = -\alpha(\nabla\vec{f})^{-1}\vec{f}$.

Gradient-Descent

An iterative method to solve nonlinear systems that uses iteration step given by $\vec{h} = -\alpha\nabla\vec{f}$.

Quasi-Newton

A variant of the Newton method that uses BFGS approximation of the Hessian $\nabla^2\vec{f}$ or its inverse $(\nabla^2\vec{f})^{-1}$, based on values from past iterations.

Gauss-Newton

A variant of the Newton method for *least-squares* problems that uses iteration step given by the minimization $\vec{h} = -\alpha(\nabla\vec{f}^T\nabla\vec{f})^{-1}(\nabla\vec{f}^T\vec{f})$, where the Hessian $\nabla^2\vec{f}$ is approximated by $\nabla\vec{f}^T\nabla\vec{f}$ with $\nabla\vec{f}$ being the Jacobian of the residuals $\vec{f}$.

Levenberg-Marquardt

A hybrid G-N and G-D method for *least-squares* problems that uses iteration step given by the minimization $\vec{h} = -\alpha(\nabla\vec{f}^T\nabla\vec{f} + \mu\vec{D})^{-1}(\nabla\vec{f}^T\vec{f})$, where *mu>0* is the damping term selecting the method G-N (small μ) or G-D (large μ), and $\vec{D} = \mathrm{diag}(\nabla\vec{f}^T\nabla\vec{f})$.

Simplex

A linear programming method (simplex method) working without using any derivatives.

Nelder-Mead

A nonlinear programming method (downhill simplex method) working without using any derivatives.

Principal-Axis

An adaptive coordinate descent method working without using any derivatives, selecting the descent direction from the Principal Component Analysis.

7.1 Stopping criteria

The `match` command will stop the iteration of the algorithm and return one of the following `status` if the corresponding criteria, *checked in this order*, is fulfilled (see also Table 14.1):

`STOPPED`

Check `env.stop == true`, i.e. termination forced by a user-defined function.

`FMIN`

Check $f \leq f_{\min}$ if $c_{\text{fail}} = 0$ or `bstra == 0`, where f is the current value of the objective function, and c_{fail} is the number of failed constraints (i.e. feasible point).

`FTOL`

Check $|\Delta f| \leq f_{\text{tol}}$ or $|\Delta f| \leq f_{\text{rtol}}\,|f|$ if $c_{\text{fail}} = 0$, where f and Δf are the current value and step of the objective function, and c_{fail} the number of failed constraints (i.e. feasible point).

`XTOL`

Check $\max(|\Delta\vec{x}| - \vec{x}_{\text{tol}}) \leq 0$ or $\max(|\Delta\vec{x}| - \vec{x}_{\text{rtol}} \circ |\vec{x}|) \leq 0$, where $\vec{x}$ and $\Delta\vec{x}$ are the current values and steps of the variables. Note that these criteria are checked even for non feasible points, i.e. $c_{\text{fail}} > 0$, as the algorithm can be trapped in a local minima that does not satisfy the constraints.

ROUNDOFF
Check $\max(|\Delta\vec{x}| - \varepsilon\,|\vec{x}|) \leq 0$ if $\vec{x}_{\text{rtol}} < \varepsilon$, where $\vec{x}$ and $\Delta\vec{x}$ are the current values and steps of the variables. The *LSopt* module returns also this status if the Jacobian is full of zeros, which is `jtol` dependent during its `jstra` reductions.

MAXCALL
Check `env.ncall >= maxcall` if `maxcall > 0`.

MAXTIME
Check `env.dtime >= maxtime` if `maxtime > 0`.

7.2 Objective function

The objective function is the key point of the `match` command, specially when tolerances are applied to it or to the constraints, or the best case strategy is changed. It is evaluated as follows:

1. Update user's `variables` with the *vector* $\vec{x}$.
2. Evaluate the *callable* `command` if defined and pass its value to the constraints.
3. Evaluate the *callable* `objective.exec` if defined and save its value f.
4. Evaluate the *callable* `equalities.exec` if defined, otherwise evaluate all the functions `equalities[].expr(cmd,env)`, and use the result to fill the *vector* $\vec{c}^{=}$.
5. Evaluate the *callable* `inequalities.exec` if defined, otherwise evaluate all the functions `inequalities[].expr(cmd,env)` and use the result to fill the *vector* $\vec{c}^{\leq}$.
6. Count the number of invalid constraints $c_{\text{fail}} = \text{card}\{|\vec{c}^{=}| > \vec{c}^{=}_{\text{tol}}\} + \text{card}\{\vec{c}^{\leq} > \vec{c}^{\leq}_{\text{tol}}\}$.
7. Calculate the *penalty* $p = \|\vec{c}\|/\|\vec{w}\|$, where $\vec{c} = \vec{w} \circ \left[\begin{smallmatrix}\vec{c}^{=} \\ \vec{c}^{\leq}\end{smallmatrix}\right]$ and $\vec{w}$ is the weights *vector* of the constraints. Set $f = p$ if the *callable* `objective.exec` is undefined.[5]
8. Save the current iteration state as the best state depending on the strategy `bstra`. The default `bstra=nil` corresponds to the last strategy

bstra	Strategy for selecting the best case of the objective function.
0	$f < f^{\text{best}}_{\text{min}}$, no feasible point check.
1	$c_{\text{fail}} \leq c^{\text{best}}_{\text{fail}}$ and $f < f^{\text{best}}_{\text{min}}$, improve both feasible point and objective.
-	$c_{\text{fail}} < c^{\text{best}}_{\text{fail}}$ or $c_{\text{fail}} = c^{\text{best}}_{\text{fail}}$ and $f < f^{\text{best}}_{\text{min}}$, improve feasible point or objective.

7.3 Derivatives

The derivatives are approximated by the finite difference methods when the selected algorithm requires them (D) and the function `objective.exec` is undefined or the attribute `grad=false`. The difficulty of the finite difference methods is to choose the small step h for the difference. The `match` command uses the *forward difference method* with a step $h = 10^{-4}\,\|\vec{h}\|$, where $\vec{h}$ is the last *iteration steps*, unless it is overridden by the user with the variable attribute `step`. In order to avoid zero step size, which would be problematic for the

[5] The LSopt module sets the values of valid inequalities to zero, i.e. $\vec{c}^{\leq} = 0$ if $\vec{c}^{\leq} \leq \vec{c}^{\leq}_{\text{tol}}$.

calculation of the Jacobian, the choice of h is a bit more subtle:

$$\frac{\partial f_j}{\partial x_i} \approx \frac{f_j(\vec{x}+h\vec{e_i}) - f_j(\vec{x})}{h} \quad ; \quad h = \begin{cases} 10^{-4}\,\|\vec{h}\| & \text{if } \|\vec{h}\| \neq 0 \\ 10^{-8}\,\|\vec{x}\| & \text{if } \|\vec{h}\| = 0 \text{ and } \|\vec{x}\| \neq 0 \\ 10^{-10} & \text{otherwise.} \end{cases}$$

Hence the approximation of the Jacobian will need an extra evaluation of the objective function per variable. If this evaluation has an heavy cost, e.g. like a `twiss` command, it is possible to approximate the Jacobian evolution by a Broyden's rank-1 update with the `broyden` attribute:

$$\vec{J}_{k+1} = \vec{J}_k + \frac{\vec{f}(\vec{x}_k+\vec{h}_k) - \vec{f}(\vec{x}_k) - \vec{J}_k\,\vec{h}_k}{\|\vec{h}_k\|^2}\,\vec{h}_k^T$$

The update of the i-th column of the Jacobian by the Broyden approximation makes sense if the angle between $\vec{h}$ and $\vec{e}_i$ is small, that is when $|\vec{h}^T\vec{e}_i| \geq \gamma\,\|\vec{h}\|$. The `match` command uses a rather pessimistic choice of $\gamma = 0.8$, which gives good performance. Nevertheless, it is advised to always check if Broyden's update saves evaluations of the objective function for your study.

8 Console output

The verbosity of the output of the `match` command on the console (e.g. terminal) is controlled by the `info` level, where the level `info=0` means a completely silent command as usual. The first verbose level `info=1` displays the *final summary* at the end of the matching, as shown in Listing 14.2 and the next level `info=2` adds *intermediate summary* for each evaluation of the objective function, as shown in Listing 14.3. The columns of these tables are self-explanatory, and the sign > on the right of the constraints marks those failing.

The bottom line of the *intermediate summary* displays in order:

- the number of evaluation of the objective function so far,
- the elapsed time in second (in square brackets) so far,
- the current objective function value,
- the current objective function step,
- the current number of constraint that failed c_{fail}.

The bottom line of the *final summary* displays the same information but for the best case found, as well as the final status returned by the `match` command. The number in square brackets right after `fbst` is the evaluation number of the best case.

The *LSopt* module adds the sign # to mark the *adjusted* variables and the sign * to mark the *rejected* variables and constraints on the right of the *intermediate summary* tables to qualify the behavior of the constraints and the variables during the optimization process. If these signs appear in the *final summary* too, it means that they were always adjusted or rejected during the matching, which is useful to tune your study e.g. by removing the useless constraints.

Listing 14.2: Match command summary output (info=1).

```
Constraints                Type        Kind        Weight     Penalty Value
-------------------------------------------------------------------------
1 IP8                      equality    beta        1          9.41469e-14
```

(continues on next page)

(continued from previous page)

```
2 IP8                      equality    beta        1          3.19744e-14
3 IP8                      equality    alfa        10         0.00000e+00
4 IP8                      equality    alfa        10         1.22125e-14
5 IP8                      equality    dx          10         5.91628e-14
6 IP8                      equality    dpx         100        1.26076e-13
7 E.DS.R8.B1               equality    beta        1          7.41881e-10
8 E.DS.R8.B1               equality    beta        1          1.00158e-09
9 E.DS.R8.B1               equality    alfa        10         4.40514e-12
10 E.DS.R8.B1              equality    alfa        10         2.23532e-11
11 E.DS.R8.B1              equality    dx          10         7.08333e-12
12 E.DS.R8.B1              equality    dpx         100        2.12877e-13
13 E.DS.R8.B1              equality    mu1         10         2.09610e-12
14 E.DS.R8.B1              equality    mu2         10         1.71063e-12

Variables                  Final Value  Init. Value  Lower Limit  Upper Limit
-------------------------------------------------------------------------------
1 kq4.l8b1                -3.35728e-03 -4.31524e-03 -8.56571e-03  0.00000e+00
2 kq5.l8b1                 4.93618e-03  5.28621e-03  0.00000e+00  8.56571e-03
3 kq6.l8b1                -5.10313e-03 -5.10286e-03 -8.56571e-03  0.00000e+00
4 kq7.l8b1                 8.05555e-03  8.25168e-03  0.00000e+00  8.56571e-03
5 kq8.l8b1                -7.51668e-03 -5.85528e-03 -8.56571e-03  0.00000e+00
6 kq9.l8b1                 7.44662e-03  7.07113e-03  0.00000e+00  8.56571e-03
7 kq10.l8b1               -6.73001e-03 -6.39311e-03 -8.56571e-03  0.00000e+00
8 kqtl11.l8b1              6.85635e-04  7.07398e-04  0.00000e+00  5.56771e-03
9 kqt12.l8b1              -2.38722e-03 -3.08650e-03 -5.56771e-03  0.00000e+00
10 kqt13.l8b1              5.55969e-03  3.78543e-03  0.00000e+00  5.56771e-03
11 kq4.r8b1                4.23719e-03  4.39728e-03  0.00000e+00  8.56571e-03
12 kq5.r8b1               -5.02348e-03 -4.21383e-03 -8.56571e-03  0.00000e+00
13 kq6.r8b1                4.18341e-03  4.05914e-03  0.00000e+00  8.56571e-03
14 kq7.r8b1               -5.48774e-03 -6.65981e-03 -8.56571e-03  0.00000e+00
15 kq8.r8b1                5.88978e-03  6.92571e-03  0.00000e+00  8.56571e-03
16 kq9.r8b1               -3.95756e-03 -7.46154e-03 -8.56571e-03  0.00000e+00
17 kq10.r8b1               7.18012e-03  7.55573e-03  0.00000e+00  8.56571e-03
18 kqtl11.r8b1            -3.99902e-03 -4.78966e-03 -5.56771e-03  0.00000e+00
19 kqt12.r8b1             -1.95221e-05 -1.74210e-03 -5.56771e-03  0.00000e+00
20 kqt13.r8b1             -2.04425e-03 -3.61438e-03 -5.56771e-03  0.00000e+00

ncall=381 [4.1s], fbst[381]=8.80207e-12, fstp=-3.13047e-08, status=FMIN.
```

Listing 14.3: Match command intermediate output (info=2).

```
 Constraints                Type        Kind        Weight     Penalty Value
-------------------------------------------------------------------------------
1 IP8                      equality    beta        1          3.10118e+00 >
2 IP8                      equality    beta        1          1.85265e+00 >
3 IP8                      equality    alfa        10         9.77591e-01 >
```

(continues on next page)

(continued from previous page)

```
4 IP8                      equality    alfa        10          8.71014e-01 >
5 IP8                      equality    dx          10          4.37803e-02 >
6 IP8                      equality    dpx         100         4.59590e-03 >
7 E.DS.R8.B1               equality    beta        1           9.32093e+01 >
8 E.DS.R8.B1               equality    beta        1           7.60213e+01 >
9 E.DS.R8.B1               equality    alfa        10          2.98722e+00 >
10 E.DS.R8.B1              equality    alfa        10          1.04758e+00 >
11 E.DS.R8.B1              equality    dx          10          7.37813e-02 >
12 E.DS.R8.B1              equality    dpx         100         6.67388e-03 >
13 E.DS.R8.B1              equality    mu1         10          7.91579e-02 >
14 E.DS.R8.B1              equality    mu2         10          6.61916e-02 >

Variables                  Curr. Value  Curr. Step   Lower Limit  Upper Limit
------------------------------------------------------------------------------
1 kq4.l8b1                -3.36997e-03 -4.81424e-04 -8.56571e-03  0.00000e+00 #
2 kq5.l8b1                 4.44028e-03  5.87400e-04  0.00000e+00  8.56571e-03
3 kq6.l8b1                -4.60121e-03 -6.57316e-04 -8.56571e-03  0.00000e+00 #
4 kq7.l8b1                 7.42273e-03  7.88826e-04  0.00000e+00  8.56571e-03
5 kq8.l8b1                -7.39347e-03  0.00000e+00 -8.56571e-03  0.00000e+00 *
6 kq9.l8b1                 7.09770e-03  2.58912e-04  0.00000e+00  8.56571e-03
7 kq10.l8b1               -5.96101e-03 -8.51573e-04 -8.56571e-03  0.00000e+00 #
8 kqtl11.l8b1              6.15659e-04  8.79512e-05  0.00000e+00  5.56771e-03 #
9 kqt12.l8b1              -2.66538e-03  0.00000e+00 -5.56771e-03  0.00000e+00 *
10 kqt13.l8b1              4.68776e-03  0.00000e+00  0.00000e+00  5.56771e-03 *
11 kq4.r8b1                4.67515e-03 -5.55795e-04  0.00000e+00  8.56571e-03 #
12 kq5.r8b1               -4.71987e-03  5.49407e-04 -8.56571e-03  0.00000e+00 #
13 kq6.r8b1                4.68747e-03 -5.54035e-04  0.00000e+00  8.56571e-03 #
14 kq7.r8b1               -5.35315e-03  4.58938e-04 -8.56571e-03  0.00000e+00 #
15 kq8.r8b1                5.77068e-03  0.00000e+00  0.00000e+00  8.56571e-03 *
16 kq9.r8b1               -4.97761e-03 -7.11087e-04 -8.56571e-03  0.00000e+00 #
17 kq10.r8b1               6.90543e-03  4.33052e-04  0.00000e+00  8.56571e-03
18 kqtl11.r8b1            -4.16758e-03 -5.95369e-04 -5.56771e-03  0.00000e+00 #
19 kqt12.r8b1             -1.57183e-03  0.00000e+00 -5.56771e-03  0.00000e+00 *
20 kqt13.r8b1             -2.57565e-03  0.00000e+00 -5.56771e-03  0.00000e+00 *

ncall=211 [2.3s], fval=8.67502e-01, fstp=-2.79653e+00, ccnt=14.
```

9 Modules

The `match` command can be extended easily with new optimizer either from external libraries or internal module, or both. The interface should be flexible and extensible enough to support new algorithms and new options with a minimal effort.

9.1 LSopt

The LSopt (Least Squares optimization) module implements custom variant of the Newton-Raphson and the Levenberg-Marquardt algorithms to solve least squares problems. Both support the options `rcond`, `bisec`, `jtol`, `jiter` and `jstra` described in the section *objective*, with the same default values. Table 14.3 lists the names of the algorithms for the attribute `method`. These algorithms cannot be used with the attribute `submethod` for the augmented algorithms of the *NLopt* module, which would not make sense as these methods support both equalities and inequalities.

Table14.3: List of supported least squares methods (LSopt).

Method	Equ	Iqu	Description
`LD_JACOBIAN`	y	y	Modified Newton-Raphson algorithm.
`LD_LMDIF`	y	y	Modified Levenberg-Marquardt algorithm.

9.2 NLopt

The NLopt (Non-Linear optimization) module provides a simple interface to the algorithms implemented in the embedded NLopt library. Table 14.4 and Table 14.5 list the names of the local and global algorithms respectively for the attribute `method`. The methods that do not support equalities (column Equ) or inequalities (column Iqu) can still be used with constraints by specifying them as the `submethod` of the AUGmented LAGrangian `method`. For details about these algorithms, please refer to the Algorithms section of its online documentation.

Table14.4: List of non-linear local methods (NLopt)

Method	Equ	Iqu	Description
Local optimizers without derivative (LN_)			
LN_BOBYQA	n	n	Bound-constrained Optimization BY Quadratic Approximations algorithm.
LN_COBYLA	y	y	Bound Constrained Optimization BY Linear Approximations algorithm.
LN_NELDERMEAD	n	n	Original Nelder-Mead algorithm.
LN_NEWUOA	n	n	Older and less efficient LN_BOBYQA.
LN_NEWUOA_BOUND	n	n	Older and less efficient LN_BOBYQA with bound constraints.
LN_PRAXIS	n	n	PRincipal-AXIS algorithm.
LN_SBPLX	n	n	Subplex algorithm, variant of Nelder-Mead.
Local optimizers with derivative (LD_)			
LD_CCSAQ	n	y	Conservative Convex Separable Approximation with Quatratic penalty.
LD_LBFGS	n	n	BFGS algorithm with low memory footprint.
LD_LBFGS_NOCEDAL	n	n	Variant from J. Nocedal of LD_LBFGS.
LD_MMA	n	y	Method of Moving Asymptotes algorithm.
LD_SLSQP	y	y	Sequential Least-Squares Quadratic Programming algorithm.
LD_TNEWTON	n	n	Inexact Truncated Newton algorithm.
LD_TNEWTON_PRECOND	n	n	Idem LD_TNEWTON with preconditioning.
LD_TNEWTON_PRECOND_RESTART	n	n	Idem LD_TNEWTON with preconditioning and steepest-descent restarting.
LD_TNEWTON_RESTART	n	n	Idem LD_TNEWTON with steepest-descent restarting.
LD_VAR1	n	n	Shifted limited-memory VARiable-metric rank-1 algorithm.
LD_VAR2	n	n	Shifted limited-memory VARiable-metric rank-2 algorithm.

Table14.5: List of supported non-linear global methods (NLopt).

Method	Equ	Iqu	Description
Global optimizers without derivative (GN_)			
GN_CRS2_LM	n	n	Variant of the Controlled Random Search algorithm with Local Mutation (mixed stochastic and genetic method).
GN_DIRECT	n	n	DIviding RECTangles algorithm (deterministic method).
GN_DIRECT_L	n	n	Idem GN_DIRECT with locally biased optimization.
GN_DIRECT_L_RAND	n	n	Idem GN_DIRECT_L with some randomization in the selection of the dimension to reduce next.
GN_DIRECT*_NOSCAL	n	n	Variants of above GN_DIRECT* without scaling the problem to a unit hypercube to preserve dimension weights.
GN_ESCH	n	n	Modified Evolutionary algorithm (genetic method).
GN_ISRES	y	y	Improved Stochastic Ranking Evolution Strategy algorithm (mixed genetic and variational method).
GN_MLSL	n	n	Multi-Level Single-Linkage algorithm (stochastic method).
GN_MLSL_LDS	n	n	Idem GN_MLSL with low-discrepancy scan sequence.
Global optimizers with derivative (GD_)			
GD_MLSL	n	n	Multi-Level Single-Linkage algorithm (stochastic method).
GD_MLSL_LDS	n	n	Idem GL_MLSL with low-discrepancy scan sequence.
GD_STOGO	n	n	Branch-and-bound algorithm (deterministic method).
GD_STOGO_RAND	n	n	Variant of GD_STOGO (deterministic and stochastic method).
AUGLAG	y	y	Augmented Lagrangian algorithm, combines objective function and nonlinear constraints into a single "penalty" function.
AUGLAG_EQ	y	n	Idem AUGLAG but handles only equality constraints and pass inequality constraints to submethod.
G_MLSL	n	n	MLSL with user-specified local algorithm using submethod.
G_MLSL_LDS	n	n	Idem G_MLSL with low-discrepancy scan sequence.

10 Examples

10.1 Matching tunes and chromaticity

The following example below shows how to match the betatron tunes of the LHC beam 1 to $q_1 = 64.295$ and $q_2 = 59.301$ using the quadrupoles strengths kqtf and kqtd, followed by the matching of the chromaticities to $dq_1 = 15$ and $dq_2 = 15$ using the main sextupole strengths ksf and ksd.

```
local lhcb1 in MADX
local twiss, match in MAD

local status, fmin, ncall = match {
  command    := twiss { sequence=lhcb1, cofind=true,
                        method=4, observe=1 },
  variables  = { rtol=1e-6, -- 1 ppm
```

(continues on next page)

(continued from previous page)

```
                { var='MADX.kqtf_b1' },
                { var='MADX.kqtd_b1' }},
  equalities = {{ expr=\t -> t.q1- 64.295, name='q1' },
                { expr=\t -> t.q2- 59.301, name='q2' }},
  objective  = { fmin=1e-10, broyden=true },
  maxcall=100, info=2
}
local status, fmin, ncall = match {
  command   := twiss { sequence=lhcb1, cofind=true, chrom=true,
                       method=4, observe=1 },
  variables  = { rtol=1e-6, -- 1 ppm
                 { var='MADX.ksf_b1' },
                 { var='MADX.ksd_b1' }},
  equalities = {{ expr= \t -> t.dq1-15, name='dq1' },
                { expr= \t -> t.dq2-15, name='dq2' }},
  objective  = { fmin=1e-8, broyden=true },
  maxcall=100, info=2
}
```

10.2 Matching interaction point

The following example hereafter shows how to squeeze the beam 1 of the LHC to $\beta^* = \mathrm{beta_ip8} \times 0.6^2$ at the IP8 while enforcing the required constraints at the interaction point and the final dispersion suppressor (i.e. at makers `"IP8"` and `"E.DS.R8.B1"`) in two iterations, using the 20 quadrupoles strengths from `kq4` to `kqt13` on left and right sides of the IP. The boundary conditions are specified by the beta0 blocks `bir8b1` for the initial conditions and `eir8b1` for the final conditions. The final summary and an instance of the intermediate summary of this `match` example are shown in Listing 14.2 and *Match command intermediate output (info=2).*.

```
local SS, ES = "S.DS.L8.B1", "E.DS.R8.B1"
lhcb1.range = SS.."/"..ES
for n=1,2 do
        beta_ip8 = beta_ip8*0.6
        local status, fmin, ncall = match {
               command := twiss { sequence=lhcb1, X0=bir8b1, method=4,␣
↪observe=1 },
               variables = { sign=true, rtol=1e-8, -- 20 variables
                { var='MADX.kq4_l8b1', name='kq4.l8b1', min=-lim2, max=lim2 },
                { var='MADX.kq5_l8b1', name='kq5.l8b1', min=-lim2, max=lim2 },
                { var='MADX.kq6_l8b1', name='kq6.l8b1', min=-lim2, max=lim2 },
                { var='MADX.kq7_l8b1', name='kq7.l8b1', min=-lim2, max=lim2 },
                { var='MADX.kq8_l8b1', name='kq8.l8b1', min=-lim2, max=lim2 },
                { var='MADX.kq9_l8b1', name='kq9.l8b1', min=-lim2, max=lim2 },
                { var='MADX.kq10_l8b1', name='kq10.l8b1', min=-lim2, max=lim2 },
```

(continues on next page)

(continued from previous page)

```
                { var='MADX.kqtl11_l8b1', name='kqtl11.l8b1', min=-lim3,␣
↪max=lim3 },
                { var='MADX.kqt12_l8b1', name='kqt12.l8b1' , min=-lim3,␣
↪max=lim3 },
                { var='MADX.kqt13_l8b1', name='kqt13.l8b1', min=-lim3, max=lim3␣
↪},
                { var='MADX.kq4_r8b1', name='kq4.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kq5_r8b1', name='kq5.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kq6_r8b1', name='kq6.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kq7_r8b1', name='kq7.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kq8_r8b1', name='kq8.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kq9_r8b1', name='kq9.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kq10_r8b1', name='kq10.r8b1', min=-lim2, max=lim2 },
                { var='MADX.kqtl11_r8b1', name='kqtl11.r8b1', min=-lim3,␣
↪max=lim3 },
                { var='MADX.kqt12_r8b1', name='kqt12.r8b1', min=-lim3, max=lim3␣
↪},
                { var='MADX.kqt13_r8b1', name='kqt13.r8b1', min=-lim3, max=lim3␣
↪},
               },
               equalities = { -- 14 equalities
                { expr=\t -> t.IP8.beta11-beta_ip8, kind='beta', name='IP8' },
                { expr=\t -> t.IP8.beta22-beta_ip8, kind='beta', name='IP8' },
                { expr=\t -> t.IP8.alfa11, kind='alfa', name='IP8' },
                { expr=\t -> t.IP8.alfa22, kind='alfa', name='IP8' },
                { expr=\t -> t.IP8.dx, kind='dx', name='IP8' },
                { expr=\t -> t.IP8.dpx, kind='dpx', name='IP8' },
                { expr=\t -> t[ES].beta11-eir8b1.beta11, kind='beta', name=ES },
                { expr=\t -> t[ES].beta22-eir8b1.beta22, kind='beta', name=ES },
                { expr=\t -> t[ES].alfa11-eir8b1.alfa11, kind='alfa', name=ES },
                { expr=\t -> t[ES].alfa22-eir8b1.alfa22, kind='alfa', name=ES },
                { expr=\t -> t[ES].dx-eir8b1.dx, kind='dx', name=ES },
                { expr=\t -> t[ES].dpx-eir8b1.dpx, kind='dpx', name=ES },
                { expr=\t -> t[ES].mu1-muxip8, kind='mu1', name=ES },
                { expr=\t -> t[ES].mu2-muyip8, kind='mu2', name=ES },
               },
               objective = { fmin=1e-10, broyden=true },
               maxcall=1000, info=2
        }
        MADX.n, MADX.tar = n, fmin

end
```

10.3 Fitting data

The following example shows how to fit data with a non-linear model using the least squares methods. The "measurements" are generated by the data function:

$$d(x) = a\sin(xf_1)\cos(xf_2), \quad \text{with} \quad a = 5, f1 = 3, f2 = 7, \text{ and } x \in [0, \pi).$$

The least squares minimization is performed by the small code below starting from the arbitrary values $a = 1$, $f_1 = 1$, and $f_2 = 1$. The `'LD_JACOBIAN'` methods finds the values $a = 5 \pm 10^{-10}$, $f_1 = 3 \pm 10^{-11}$, and $f_2 = 7 \pm 10^{-11}$ in 2574 iterations and 0.1,s. The `'LD_LMDIF'` method finds similar values in 2539 iterations. The data and the model are plotted in the Fig. 14.1.

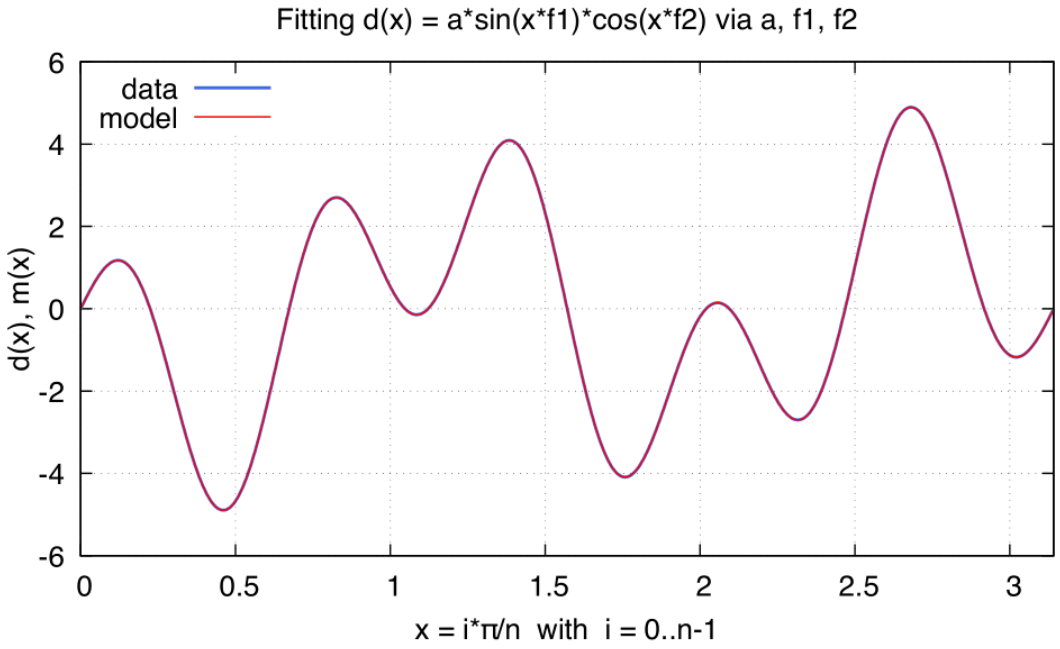

Figure14.1: Fitting data using the Jacobian or Levenberg-Marquardt methods.}

```
local n, k, a, f1, f2 = 1000, pi/1000, 5, 3, 7
local d = vector(n):seq():map \i -> a*sin(i*k*f1)*cos(i*k*f2) -- data
if noise then d=d:map \x -> x+randtn(noise) end -- add noise if any
local m, p = vector(n), { a=1, f1=1, f2=1 } -- model parameters
local status, fmin, ncall = match {
        command       := m:seq():map \i -> p.a*sin(i*k*p.f1)*cos(i*k*p.f2),
        variables     = { { var='p.a' },
                          { var='p.f1' },
                          { var='p.f2' }, min=1, max=10 },
        equalities    = { { expr=\m -> ((d-m):norm()) } },
        objective     = { fmin=1e-9, bisec=noise and 5 },
        maxcall=3000, info=1
}
```

The same least squares minimization can be achieved on noisy data by adding a gaussian RNG truncated at 2σ to the data generator, i.e.:literal:*noise*=2, and by increasing the attribute `bisec=5`. Of course, the penalty tolerance `fmin` must be moved to variables tolerance `tol` or `rtol`. The `'LD_JACOBIAN'` methods finds the values $a = 4.98470$, $f_1 = 3.00369$, and $f_2 = 6.99932$ in 704 iterations (404 for `'LD_LMDIF'`). The data and the model are plotted in Fig. 14.2.

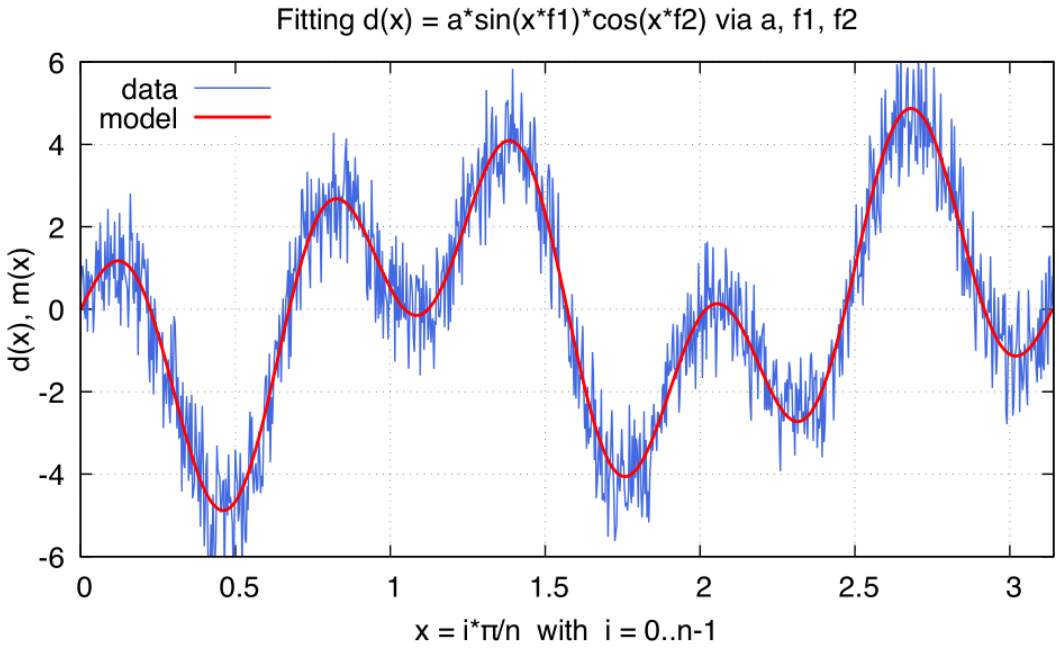

Figure14.2: Fitting data with noise using Jacobian or Levenberg-Marquardt methods.

10.4 Fitting data with derivatives

The following example shows how to fit data with a non-linear model and its derivatives using the least squares methods. The least squares minimization is performed by the small code below starting from the arbitrary values $v = 0.9$ and $k = 0.2$. The `'LD_JACOBIAN'` methods finds the values $v = 0.362 \pm 10^{-3}$ and $k = 0.556 \pm 10^{-3}$ in 6 iterations. The `'LD_LMDIF'` method finds similar values in 6 iterations too. The data (points) and the model (curve) are plotted in the Fig. 14.3, where the latter has been smoothed using cubic splines.

```
local x = vector{0.038, 0.194, 0.425, 0.626 , 1.253 , 2.500 , 3.740 }
local y = vector{0.050, 0.127, 0.094, 0.2122, 0.2729, 0.2665, 0.3317}
local p = { v=0.9, k=0.2 }
local n = #x
local function eqfun (_, r, jac)
        local v, k in p
        for i=1,n do
                r[i] = y[i] - v*x[i]/(k+x[i])
                jac[2*i-1] = -x[i]/(k+x[i])
```

(continues on next page)

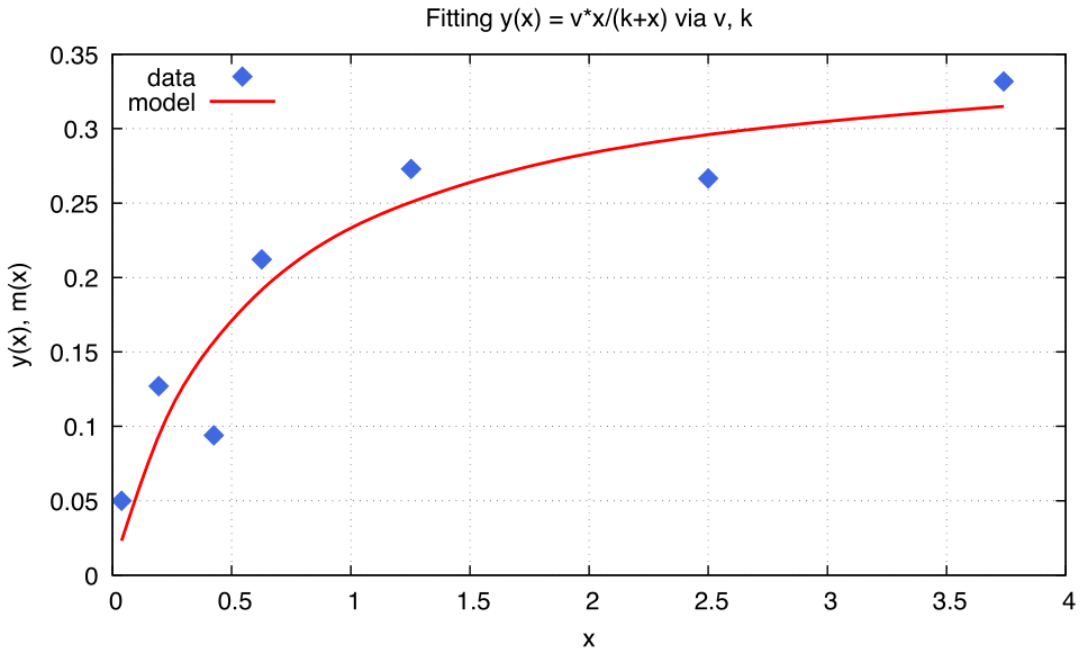

Figure14.3: Fitting data with derivatives using the Jacobian or Levenberg-Marquardt methods.

(continued from previous page)

```
                jac[2*i] = v*x[i]/(k+x[i])^2
        end
end
local status, fmin, ncall = match {
        variables       = { tol=5e-3, min=0.1, max=2,
                            { var='p.v' },
                            { var='p.k' } },
        equalities      = { nequ=n, exec=eqfun, disp=false },
        maxcall=20
```

10.5 Minimizing function

The following example[6] hereafter shows how to find the minimum of the function:

$$\min_{\vec{x}\in\mathbb{R}^2} \sqrt{x_2}, \quad \text{subject to the constraints} \quad \begin{cases} x_2 \geq 0, \\ x_2 \geq (a_1 x_1 + b_1)^3, \\ x_2 \geq (a_2 x_1 + b_2)^3, \end{cases}$$

for the parameters $a_1 = 2, b_1 = 0, a_2 = -1$ and $b_2 = 1$. The minimum of the function is $f_{\min} = \sqrt{\frac{8}{27}}$ at the point $\vec{x} = (\frac{1}{3}, \frac{8}{27})$, and found by the method `LD_MMA` in 11 evaluations for a relative tolerance of 10^{-4} on the variables, starting at the arbitrary point $\vec{x}_0 = (1.234, 5.678)$.

[6] This example is taken from the NLopt documentation.

```
local function testFuncFn (x, grd)
        if grd then x:fill{ 0, 0.5/sqrt(x[2]) } end
        return sqrt(x[2])
end
local function testFuncLe (x, r, jac)
        if jac then jac:fill{ 24*x[1]^2, -1, -3*(1-x[1])^2, -1 } end
        r:fill{ 8*x[1]^3-x[2], (1-x[1])^3-x[2] }
end
local x = vector{1.234, 5.678} -- start point
local status, fmin, ncall = match {
        variables      = { rtol=1e-4,
                           { var='x[1]', min=-inf },
                           { var='x[2]', min=0   } },
        inequalities   = { exec=testFuncLe, nequ=2, tol=1e-8 },
        objective      = { exec=testFuncFn, method='LD_MMA' },
        maxcall=100, info=2
}
```

This example can also be solved with least squares methods, where the `LD_JACOBIAN` method finds the minimum in 8 iterations with a precision of $\pm 10^{-16}$, and the `LD_LMDIF` method finds the minimum in 10 iterations with a precision of $\pm 10^{-11}$.

Chapter 15. Correct

The `correct` command (i.e. orbit correction) provides a simple interface to compute the orbit steering correction and setup the kickers of the sequences from the analysis of their `track` and `twiss` mtables.

Listing 15.1: Synopsis of the `correct` command with default setup.

```
mlst = correct {
        sequence=nil,    -- sequence(s) (required)
        range=nil,       -- sequence(s) range(s) (or sequence.range)
        title=nil,       -- title of mtable (default seq.name)
        model=nil,       -- mtable(s) with twiss functions (required)
        orbit=nil,       -- mtable(s) with measured orbit(s), or use model
        target=nil,      -- mtable(s) with target orbit(s), or zero orbit
        kind='ring',     -- 'line' or 'ring'
        plane='xy',      -- 'x', 'y' or 'xy'
        method='micado',-- 'LSQ', 'SVD' or 'MICADO'
        ncor=0,          -- number of correctors to consider by method, 0=all
        tol=1e-5,        -- rms tolerance on the orbit
        units=1,         -- units in [m] of the orbit
        corcnd=false,    -- precond of correctors using 'svdcnd' or 'pcacnd'
        corcut=0,        -- value to theshold singular values in precond
        cortol=0,        -- value to theshold correctors in svdcnd
        corset=true,     -- update correctors correction strengths
        monon=false,     -- fraction (0<?<=1) of randomly available monitors
        moncut=false,    -- cut monitors above moncut sigmas
        monerr=false,    -- 1:use mrex and mrey alignment errors of monitors
                         -- 2:use msex and msey scaling errors of monitors
        info=nil,        -- information level (output on terminal)
        debug=nil,       -- debug information level (output on terminal)
}
```

1 Command synopsis

The `correct` command format is summarized in Listing 15.1, including the default setup of the attributes. The `correct` command supports the following attributes:

sequence

The *sequence* (or a list of *sequence*) to analyze. (no default, required).

Example: `sequence = lhcb1`.

range

A *range* (or a list of *range*) specifying the span of the sequence to analyze. If no range is provided, the command looks for a range attached to the sequence, i.e. the attribute `seq.range`. (default: `nil`).

Example: `range = "S.DS.L8.B1/E.DS.R8.B1"`.

title

A *string* specifying the title of the *mtable*. If no title is provided, the command looks for the name of

the sequence, i.e. the attribute `seq.name`. (default: `nil`).

Example: `title = "Correct orbit around IP5"`.

model

A *mtable* (or a list of *mtable*) providing `twiss`-like information, e.g. elements, orbits and optical functions, of the corresponding sequences. (no default, required).

Example: `model = twmtbl`.

orbit

A *mtable* (or a list of *mtable*) providing `track`-like information, e.g. elements and measured orbits, of the corresponding sequences. If this attribute is `nil`, the model orbit is used. (default: `nil`).

Example: `orbit = tkmtbl`.

target

A *mtable* (or a list of *mtable*) providing `track`-like information, e.g. elements and target orbits, of the corresponding sequences. If this attribute is `nil`, the design orbit is used. (default: `nil`).

Example: `target = tgmtbl`.

kind

A *string* specifying the kind of correction to apply among `line` or `ring`. The kind `line` takes care of the causality between monitors, correctors and sequences directions, while the kind `ring` considers the system as periodic. (default: `'ring'`).

Example: `kind = 'line'`.

plane

A *string* specifying the plane to correct among `x`, , `y` and `xy`. (default: `'xy'`).

Example: `plane = 'x'`.

method

A *string* specifying the method to use for correcting the orbit among `LSQ`, `SVD` or `micado`. These methods correspond to the solver used from the *linear algebra* module to find the orbit correction, namely `solve`, `ssolve` or `nsolve`. (default: `'micado'`).

Example: `method = 'svd'`.

ncor

A *number* specifying the number of correctors to consider with the method `micado`, zero meaning all available correctors. (default: `0`).

Example: `ncor = 4`.

tol

A *number* specifying the rms tolerance on the residuals for the orbit correction. (default: 1e-6).

Example: `tol = 1e-6`.

unit

A *number* specifying the unit of the `orbit` and `target` coordinates. (default: `1` [m]).

Example: `units = 1e-3` [m], i.e. [mm].

corcnd

A *logical* or a *string* specifying the method to use among `svdcnd` and `pcacnd` from the *linear algebra* module for the preconditioning of the system. A `true` value corresponds to . (default: `false`).

Example: `corcnd = 'pcacnd'`.

corcut

A *number* specifying the thresholds for the singular values to pass to the `svdcnd` and `pcacnd` method

for the preconditioning of the system. (default: `0`).

Example: `cortol = 1e-6`.

cortol

A *number* specifying the thresholds for the correctors to pass to the `svdcnd` method for the preconditioning of the system. (default: `0`).

Example: `cortol = 1e-8`.

corset

A *logical* specifying to update the correctors strengths for the corrected orbit. (default: `true`).

Example: `corset = false`.

monon

A *number* specifying a fraction of available monitors selected from a uniform RNG. (default: `false`).

Example: `monon = 0.8`, keep 80% of the monitors.

moncut

A *number* specifying a threshold in number of sigma to cut monitor considered as outliers. (default: `false`).

Example: `moncut = 2`, cut monitors above 2σ.

monerr

A *number* in `0..3` specifying the type of monitor reading errors to consider: `1` use scaling errors `msex` and `msey`, `2` use alignment errors `mrex`, `mrey` and `dpsi`, `3` use both. (default: `false`).

Example: `monerr = 3`.

info

A *number* specifying the information level to control the verbosity of the output on the console. (default: `nil`).

Example: `info = 2`.

debug

A *number*specifying the debug level to perform extra assertions and to control the verbosity of the output on the console. (default: `nil`).

Example: `debug = 2`.

The `correct` command returns the following object:

`mlst`

A *mtable* (or a list of *mtable*) corresponding to the TFS table of the `correct` command. It is a list when multiple sequences are corrected together.

2 Correct mtable

The `correct` command returns a *mtable* where the information described hereafter is the default list of fields written to the TFS files.[1]

The header of the *mtable* contains the fields in the default order:

name

The name of the command that created the *mtable*, e.g. `"correct"`.

[1] The output of mtable in TFS files can be fully customized by the user.

type
The type of the *mtable*, i.e. `"correct"`.

title
The value of the command attribute `title`.

origin
The origin of the application that created the *mtable*, e.g. `"MAD 1.0.0 OSX 64"`.

date
The date of the creation of the *mtable*, e.g. `"27/05/20"`.

time
The time of the creation of the *mtable*, e.g. `"19:18:36"`.

refcol
The reference *column* for the *mtable* dictionnary, e.g. `"name"`.

range
The value of the command attribute `range`.[2]

__seq
The *sequence* from the command attribute `sequence`.[3] .. _ref.track.mtbl1}:

The core of the *mtable* contains the columns in the default order:

name
The name of the element.

kind
The kind of the element.

s
The s-position at the end of the element slice.

l
The length from the start of the element to the end of the element slice.

x_old
The local coordinate x at the s-position before correction.

y_old
The local coordinate y at the s-position before correction.

x
The predicted local coordinate x at the s-position after correction.

y
The predicted local coordinate y at the s-position after correction.

rx
The predicted local residual r_x at the s-position after correction.

ry
The predicted local residual r_y at the s-position after correction.

hkick_old
The local horizontal kick at the s-position before correction.

[2] This field is not saved in the TFS table by default.
[3] Fields and columns starting with two underscores are protected data and never saved to TFS files.label{ref:track:mtbl1

vkick_old
The local vertical kick at the s-position before correction.

hkick
The predicted local horizontal kick at the s-position after correction.

vkick
The predicted local vertical kick at the s-position after correction.

shared
A *logical* indicating if the element is shared with another sequence.

eidx
The index of the element in the sequence.

Note that `correct` does not take into account the particles and damaps `id`s present in the (augmented) `track` *mtable*, hence the provided tables should contain single particle or damap information.

3 Examples

Chapter 16. Emit

This command is not yet implemented in MAD. It will probably be implemented as a layer on top of the Twiss and Match commands.

Chapter 17. Plot

The `plot` command provides a simple interface to the Gnuplot application. The Gnuplot release 5.2 or higher must be installed and visible in the user `PATH` by MAD to be able to run this command.

1 Command synopsis

Listing 17.1: Synopsis of the `plot` command with default setup.

```
cmd = plot {
        sid                     = 1,     -- stream id 1 <= n <= 25 (Gnuplot␣
↪instances)
        data            = nil,   -- { x=tbl.x, y=vec } (precedence over table)
        table           = nil,   -- mtable
        tablerange      = nil,   -- mtable range (default table.range)
        sequence        = nil,   -- seq | { seq1, seq2, ...,} | "keep"
        range           = nil,   -- sequence range (default sequence.range)
        name            = nil,   -- (default table.title)
        date            = nil,   -- (default table.date)
        time            = nil,   -- (default table.time)
        output          = nil,   -- "filename" -> pdf | number -> wid
        scrdump         = nil,   -- "filename"
        survey-attributes,
        windows-attributes,
        layout-attributes,
        labels-attributes,
        axis-attributes,
        ranges-attributes,
        data-attributes,
        plots-attributes,
        custom-attributes,
        info            =nil,   -- information level (output on terminal)
        debug           =nil,   -- debug information level (output on terminal)
}
```

The `plot` command format is summarized in Listing 17.1, including the default setup of the attributes. The `plot` command supports the following attributes:

info
: A *number* specifying the information level to control the verbosity of the output on the console. (default: `nil`). Example: `info = 2`.

debug
: A *number* specifying the debug level to perform extra assertions and to control the verbosity of the output on the console. (default: `nil`). Example: `debug = 2`.

The `plot` command returns itself.

Part III

PHYSICS

Chapter 18. Introduction

1 Local reference system

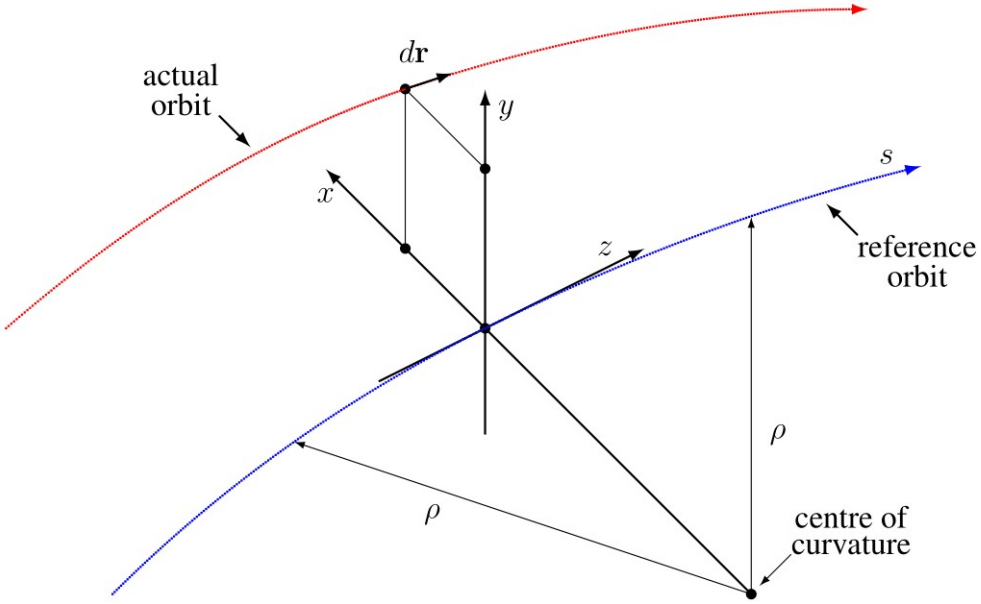

Figure18.1: Local Reference System

2 Global reference system

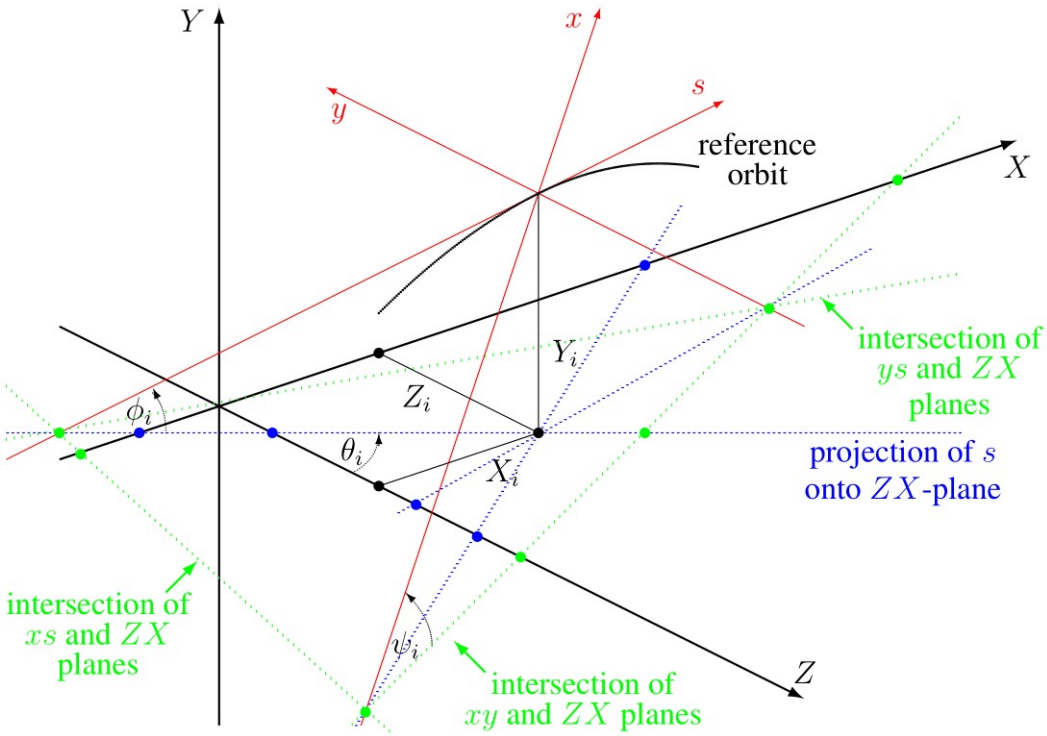

Figure18.2: Global Reference System showing the global Cartesian system (X, Y, Z) in black and the local reference system (x, y, s) in red after translation (X_i, Y_i, Z_i) and rotation $(\theta_i, \phi_i, \psi_i)$. The projections of the local reference system axes onto the horizontal ZX plane of the Cartesian system are figured with blue dashed lines. The intersections of planes ys, xy and xs of the local reference system with the horizontal ZX plane of the Cartesian system are figured in green dashed lines.

Chapter 19. Geometric Maps

Chapter 20. Dynamic Maps

Chapter 21. Integrators

Chapter 22. Orbit

1 Closed Orbit

Chapter 23. Optics

Chapter 24. Normal Forms

Chapter 25. Misalignments

Chapter 26. Aperture

Chapter 27. Radiation

Part IV

MODULES

Chapter 28. Types

This chapter describes some types identification and concepts setup defined by the module `MAD.typeid` and `MAD._C` (C API). The module `typeid` is extended by types from other modules on load like e.g. *is_range*, *is_complex*, *is_matrix*, *is_tpsa*, etc...

1 Typeids

All the functions listed hereafter return only `true` or `false` when identifying types.

1.1 Primitive Types

The following table shows the functions for identifying the primitive type of LuaJIT, i.e. using `type(a) == 'the_type'`

Functions	Return `true` if `a`
`is_nil(a)`	is a *nil*
`is_boolean(a)`	is a *boolean*
`is_number(a)`	is a *number*
`is_string(a)`	is a *string*
`is_function(a)`	is a *function*
`is_table(a)`	is a *table*
`is_userdata(a)`	is a *userdata*
`is_coroutine(a)`	is a *thread*[1]
`is_cdata(a)`	is a *cdata*

1.2 Extended Types

The following table shows the functions for identifying the extended types, which are primitive types with some extensions, specializations or value ranges.

[1] The Lua "threads" are user-level non-preemptive threads also named coroutines.

Functions	Return `true` if `a`
`is_nan(a)`	is `nan` (Not a Number)
`is_true(a)`	is `true`
`is_false(a)`	is `false`
`is_logical(a)`	is a *boolean* or `nil`
`is_finite(a)`	is a *number* with $\|a\| < \infty$
`is_infinite(a)`	is a *number* with $\|a\| = \infty$
`is_positive(a)`	is a *number* with $a > 0$
`is_negative(a)`	is a *number* with $a < 0$
`is_zpositive(a)`	is a *number* with $a \geq 0$
`is_znegative(a)`	is a *number* with $a \leq 0$
`is_nonzero(a)`	is a *number* with $a \neq 0$
`is_integer(a)`	is a *number* with $-2^{52} \leq a \leq 2^{52}$ and no fractional part
`is_int32(a)`	is a *number* with $-2^{31} \leq a < 2^{31}$ and no fractional part
`is_natural(a)`	is an *integer* with $a \geq 0$
`is_even(a)`	is an even *integer*
`is_odd(a)`	is an odd *integer*
`is_decimal(a)`	is not an *integer*
`is_emptystring(a)`	is a *string* with `#a == 0`
`is_identifier(a)`	is a *string* with valid identifier characters, i.e. `%s*[_%a][_%w]*%s*`
`is_rawtable(a)`	is a *table* with no metatable
`is_emptytable(a)`	is a *table* with no element
`is_file(a)`	is a *userdata* with `io.type(a) ~= nil`
`is_openfile(a)`	is a *userdata* with `io.type(a) == 'file'`
`is_closedfile(a)`	is a *userdata* with `io.type(a) == 'closed file'`
`is_emptyfile(a)`	is an open *file* with some content

2 Concepts

Concepts are an extention of types looking at their behavior. The concepts are more based on supported metamethods (or methods) than on the types themself and their valid range of values.

Functions	Return `true` if `a`
`is_value(a)`	is a *nil*, a *boolean*, a *number* or a *string*
`is_reference(a)`	is not a *value*
`is_empty(a)`	is a *mappable* and 1st iteration returns `nil`
`is_lengthable(a)`	supports operation `#a`
`is_iterable(a)`	supports operation `ipairs(a)`
`is_mappable(a)`	supports operation `pairs(a)`
`is_indexable(a)`	supports operation `a[?]`
`is_extendable(a)`	supports operation `a[]=?`
`is_callable(a)`	supports operation `a()`
`is_equalable(a)`	supports operation `a == ?`
`is_orderable(a)`	supports operation `a < ?`
`is_concatenable(a)`	supports operation `a .. ?`
`is_negatable(a)`	supports operation `-a`
`is_addable(a)`	supports operation `a + ?`
`is_subtractable(a)`	supports operation `a - ?`
`is_multipliable(a)`	supports operation `a * ?`
`is_dividable(a)`	supports operation `a / ?`
`is_modulable(a)`	supports operation `a % ?`
`is_powerable(a)`	supports operation `a ^ ?`
`is_copiable(a)`	supports metamethod `__copy()`
`is_sameable(a)`	supports metamethod `__same()`
`is_tablable(a)`	supports metamethod `__totable()`
`is_stringable(a)`	supports metamethod `__tostring()`
`is_mutable(a)`	defines metamethod `__metatable()`
`is_restricted(a)`	has metamethods for restriction, see *wrestrict()*
`is_protected(a)`	has metamethods for protection, see *wprotect()*
`is_deferred(a)`	has metamethods for deferred expressions, see *deferred()*
`is_same(a,b)`	has the same type and metatable as `b`

The functions in the following table are complementary to concepts and usually used to prevent an error during concepts checks.

Functions	Return `true` if
`has_member(a,b)`	`a[b]` is not `nil`
`has_method(a,f)`	`a[f]` is a *callable*
`has_metamethod(a,f)`	metamethod `f` is defined
`has_metatable(a)`	`a` has a metatable

`is_metaname`(*a*)

Returns `true` if the *string* `a` is a valid metamethod name, `false` otherwise.

`get_metatable`(*a*)

Returns the metatable of `a` even if `a` is a *cdata*, which is not the case of `getmetatable()`.

get_metamethod(*a*, *f*)

Returns the metamethod (or method) `f` of `a` even if `a` is a *cdata* and `f` is only reachable through the metatable, or `nil`.

2.1 Setting Concepts

`typeid.`**concept**

The *table* `concept` contains the lists of concepts that can be passed to the function `set_concept` to prevent the use of their associated metamethods. The concepts can be combined together by adding them, e.g. `not_comparable = not_equalable + not_orderable`.

Concepts	Associated metamethods
`not_lengthable`	`__len`
`not_iterable`	`__ipairs`
`not_mappable`	`__ipairs` and `__pairs`
`not_scannable`	`__len`, `__ipairs` and `__pairs`
`not_indexable`	`__index`
`not_extendable`	`__newindex`
`not_callable`	`__call`
`not_equalable`	`__eq`
`not_orderable`	`__lt` and `__le`
`not_comparable`	`__eq`, `__lt` and `__le`
`not_concatenable`	`__concat`
`not_copiable`	`__copy` and `__same`
`not_tablable`	`__totable`
`not_stringable`	`__tostring`
`not_mutable`	`__metatable` and `__newindex`
`not_negatable`	`__unm`
`not_addable`	`__add`
`not_subtractable`	`__sub`
`not_additive`	`__add` and `__sub`
`not_multipliable`	`__mul`
`not_dividable`	`__div`
`not_multiplicative`	`__mul` and `__div`
`not_modulable`	`__mod`
`not_powerable`	`__pow`

set_concept(*mt*, *concepts*, *strict_*)

Return the metatable `mt` after setting the metamethods associated to the combination of concepts set in `concepts` to prevent their use. The concepts can be combined together by adding them, e.g. `not_comparable = not_equalable + not_orderable`. Metamethods can be overridden if `strict = false`, otherwise the overload is silently discarded. If `concepts` requires *iterable* but not *mappable* then `pairs` is equivalent to `ipairs`.

wrestrict(*a*)

Return a proxy for a which behaves like a, except that it prevents existing indexes from being modified while allowing new ones to be created, i.e. a is *extendable*.

wprotect(*a*)

Return a proxy for a which behaves like a, except that it prevents existing indexes from being modified and does not allow new ones to be created, i.e. a is *readonly*.

wunprotect(*a*)

Return a from the proxy, i.e. expect a restricted or a protected a.

deferred(*a*)

Return a proxy for a which behaves like a except that elements of type *function* will be considered as deferred expressions and evaluated on read, i.e. returning their results in their stead.

3 C Type Sizes

The following table lists the constants holding the size of the C types used by common *cdata* like complex, matrices or TPSA. See section on *C API* for the description for those C types.

C types sizes	C types
ctsz_log	*log_t*
ctsz_idx	*idx_t*
ctsz_ssz	*ssz_t*
ctsz_dbl	*num_t*
ctsz_cpx	*cpx_t*
ctsz_str	*str_t*
ctsz_ptr	*ptr_t*

4 C API

type **log_t**

The *logical* type aliasing *_Bool*, i.e. boolean, that holds TRUE or FALSE.

type **idx_t**

The *index* type aliasing *int32_t*, i.e. signed 32-bit integer, that holds signed indexes in the range $[-2^{31}, 2^{31}-1]$.

type **ssz_t**

The *size* type aliasing *int32_t*, i.e. signed 32-bit integer, that holds signed sizes in the range $[-2^{31}, 2^{31}-1]$.

type **num_t**

The *number* type aliasing *double*, i.e. double precision 64-bit floating point numbers, that holds double-precision normalized number in IEC 60559 in the approximative range $\{-\infty\} \cup$

$[-\text{huge}, -\text{tiny}] \cup \{0\} \cup [\text{tiny}, \text{huge}] \cup \{\infty\}$ where huge $\approx 10^{308}$ and tiny $\approx 10^{-308}$. See `MAD.constant.huge` and `MAD.constant.tiny` for precise values.

type **`cpx_t`**

The *complex* type aliasing *double _Complex*, i.e. two double precision 64-bit floating point numbers, that holds double-precision normalized number in IEC 60559.

type **`str_t`**

The *string* type aliasing *const char**, i.e. pointer to a readonly null-terminated array of characters.

type **`ptr_t`**

The *pointer* type aliasing *const void**, i.e. pointer to readonly memory of unknown/any type.

Chapter 29. Constants

This chapter describes some constants uniquely defined as macros in the C header `mad_cst.h` and available from modules `MAD.constant` and `MAD._C` (C API) as floating point double precision variables.

1 Numerical Constants

These numerical constants are provided by the system libraries. Note that the constant `huge` differs from `math.huge`, which corresponds in fact to `inf`.

MAD constants	C macros	C constants	Values
`eps`	`DBL_EPSILON`	`mad_cst_EPS`	Smallest representable step 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`	`NAN`	`mad_cst_NAN`	Canonical NaN[1], $0/0$

2 Mathematical Constants

This section describes some mathematical constants uniquely defined as macros in the C header `mad_cst.h` and available from C and MAD modules as floating point double precision variables. If these mathematical constants are already provided by the system libraries, they will be used instead of their local definitions.

[1] Canonical NaN bit patterns may differ between MAD and C for the mantissa, but both should exibit the same behavior.

MAD constants	C macros	C constants	Values
e	M_E	mad_cst_E	e
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	$2/\sqrt{\pi}$
rad2deg	M_RAD2DEG	mad_cst_RAD2DEG	$180/\pi$
deg2rad	M_DEG2RAD	mad_cst_DEG2RAD	$\pi/180$

3 Physical Constants

This section describes some physical constants uniquely defined as macros in the C header `mad_cst.h` and available from C and MAD modules as floating point double precision variables.

MAD constants	C macros	C constants	Values
minlen	P_MINLEN	mad_cst_MINLEN	Min length tolerance, default 10^{-10} in **[m]**
minang	P_MINANG	mad_cst_MINANG	Min angle tolerance, default 10^{-10} in **[1/m]**
minstr	P_MINSTR	mad_cst_MINSTR	Min strength tolerance, default 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, $m_u\,c^2$ in **[GeV]**
emass	P_EMASS	mad_cst_EMASS	Electron mass, $m_e\,c^2$ in **[GeV]**
pmass	P_PMASS	mad_cst_PMASS	Proton mass, $m_p\,c^2$ in **[GeV]**
nmass	P_NMASS	mad_cst_NMASS	Neutron mass, $m_n\,c^2$ in **[GeV]**
mumass	P_MUMASS	mad_cst_MUMASS	Muon mass, $m_\mu\,c^2$ in **[GeV]**
deumass	P_DEUMASS	mad_cst_DEUMASS	Deuteron mass, $m_d\,c^2$ 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 30. Functions

This chapter describes some functions provided by the modules `MAD.gmath` and `MAD.gfunc`.

The module `gmath` extends the standard LUA module `math` with *generic* functions working on any types that support the methods with the same names. For example, the code `gmath.sin(a)` will call `math.sin(a)` if `a` is a *number*, otherwise it will call the method `a:sin()`, i.e. delegate the invocation to `a`. This is how MAD-NG handles several types like *numbers*, *complex* number and *TPSA* within a single *polymorphic* code that expects scalar-like behavior.

The module `gfunc` provides useful functions to help dealing with operators as functions and to manipulate functions in a functional way[1].

1 Mathematical Functions

1.1 Generic Real-like Functions

Real-like generic functions forward the call to the method of the same name from the first argument when the latter is not a *number*. The optional argument `r_` represents a destination placeholder for results with reference semantic, i.e. avoiding memory allocation, which is ignored by results with value semantic. The C functions column lists the C implementation used when the argument is a *number* and the implementation does not rely on the standard `math` module but on functions provided with MAD-NG or by the standard math library described in the C Programming Language Standard [ISOC99].

Functions	Return values	C functions
`abs(x,r_)`	$\lvert x\rvert$	
`acos(x,r_)`	$\cos^{-1} x$	
`acosh(x,r_)`	$\cosh^{-1} x$	`acosh()`
`acot(x,r_)`	$\cot^{-1} x$	
`acoth(x,r_)`	$\coth^{-1} x$	`atanh()`
`asin(x,r_)`	$\sin^{-1} x$	
`asinc(x,r_)`	$\frac{\sin^{-1} x}{x}$	*mad_num_asinc()*
`asinh(x,r_)`	$\sinh^{-1} x$	`asinh()`
`asinhc(x,r_)`	$\frac{\sinh^{-1} x}{x}$	*mad_num_asinhc()*
`atan(x,r_)`	$\tan^{-1} x$	
`atan2(x,y,r_)`	$\tan^{-1} \frac{x}{y}$	
`atanh(x,r_)`	$\tanh^{-1} x$	`atanh()`
`ceil(x,r_)`	$\lceil x\rceil$	
`cos(x,r_)`	$\cos x$	
`cosh(x,r_)`	$\cosh x$	
`cot(x,r_)`	$\cot x$	

continues on next page

[1] For *true* Functional Programming, see the module `MAD.lfun`, a binding of the LuaFun library adapted to the ecosystem of MAD-NG.

Table 30.1 – continued from previous page

Functions	Return values	C functions
`coth(x,r_)`	$\coth x$	
`exp(x,r_)`	$\exp x$	
`floor(x,r_)`	$\lfloor x \rfloor$	
`frac(x,r_)`	$x - \mathrm{trunc}(x)$	
`hypot(x,y,r_)`	$\sqrt{x^2 + y^2}$	`hypot()`
`hypot3(x,y,z,r_)`	$\sqrt{x^2 + y^2 + z^2}$	`hypot()`
`inv(x,v_,r_)`[2]	$\frac{v}{x}$	
`invsqrt(x,v_,r_)`Page 182, 2	$\frac{v}{\sqrt{x}}$	
`lgamma(x,tol_,r_)`	$\ln \lvert\Gamma(x)\rvert$	`lgamma()`
`log(x,r_)`	$\log x$	
`log10(x,r_)`	$\log_{10} x$	
`powi(x,n,r_)`	x^n	*mad_num_powi()*
`round(x,r_)`	$\lfloor x \rceil$	`round()`
`sign(x)`	$-1, 0$ or 1	*mad_num_sign()*[3]
`sign1(x)`	-1 or 1	*mad_num_sign1()*[3]
`sin(x,r_)`	$\sin x$	
`sinc(x,r_)`	$\frac{\sin x}{x}$	*mad_num_sinc()*
`sinh(x,r_)`	$\sinh x$	
`sinhc(x,r_)`	$\frac{\sinh x}{x}$	*mad_num_sinhc()*
`sqrt(x,r_)`	$\sqrt{x}$	
`tan(x,r_)`	$\tan x$	
`tanh(x,r_)`	$\tanh x$	
`tgamma(x,tol_,r_)`	$\Gamma(x)$	`tgamma()`
`trunc(x,r_)`	$\lfloor x \rfloor, x \geq 0; \lceil x \rceil, x < 0$	
`unit(x,r_)`	$\frac{x}{\lvert x \rvert}$	

1.2 Generic Complex-like Functions

Complex-like generic functions forward the call to the method of the same name from the first argument when the latter is not a *number*, otherwise it implements a real-like compatibility layer using the equivalent representation $z = x + 0i$. The optional argument `r_` represents a destination for results with reference semantic, i.e. avoiding memory allocation, which is ignored by results with value semantic.

[2] Default: `v_ = 1`.

[3] Sign and sign1 functions take care of special cases like ± 0, $\pm$inf and $\pm$NaN.

Functions	Return values
`cabs(z,r_)`	$\lvert z\rvert$
`carg(z,r_)`	$\arg z$
`conj(z,r_)`	z^*
`cplx(x,y,r_)`	$x + i\,y$
`fabs(z,r_)`	$\lvert\Re(z)\rvert + i\,\lvert\Im(z)\rvert$
`imag(z,r_)`	$\Im(z)$
`polar(z,r_)`	$\lvert z\rvert\, e^{i \arg z}$
`proj(z,r_)`	$\mathrm{proj}(z)$
`real(z,r_)`	$\Re(z)$
`rect(z,r_)`	$\Re(z)\cos\Im(z) + i\,\Re(z)\sin\Im(z)$
`reim(z,re_,im_)`	$\Re(z), \Im(z)$

1.3 Generic Vector-like Functions

Vector-like functions (also known as MapFold or MapReduce) are functions useful when used as high-order functions passed to methods like `:map2()`, `:foldl()` (fold left) or `:foldr()` (fold right) of containers like lists, vectors and matrices.

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)`	$\max(\lvert x\rvert, \lvert y\rvert)$
`sumsqrl(x,y)`	$x + y^2$
`sumabsl(x,y)`	$x + \lvert y\rvert$
`minabsl(x,y)`	$\min(x, \lvert y\rvert)$
`maxabsl(x,y)`	$\max(x, \lvert y\rvert)$
`sumsqrr(x,y)`	$x^2 + y$
`sumabsr(x,y)`	$\lvert x\rvert + y$
`minabsr(x,y)`	$\min(\lvert x\rvert, y)$
`maxabsr(x,y)`	$\max(\lvert x\rvert, y)$

1.4 Generic Error-like Functions

Error-like generic functions forward the call to the method of the same name from the first argument when the latter is not a *number*, otherwise it calls C wrappers to the corresponding functions from the Faddeeva library from the MIT (see `mad_num.c`). The optional argument `r_` represents a destination for results with reference semantic, i.e. avoiding memory allocation, which is ignored by results with value semantic.

Functions	Return values	C functions
`erf(z,rtol_,r_)`	$\frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$	*mad_num_erf()*
`erfc(z,rtol_,r_)`	$1-\operatorname{erf}(z)$	*mad_num_erfc()*
`erfi(z,rtol_,r_)`	$-i\operatorname{erf}(iz)$	*mad_num_erfi()*
`erfcx(z,rtol_,r_)`	$e^{z^2}\operatorname{erfc}(z)$	*mad_num_erfcx()*
`wf(z,rtol_,r_)`	$e^{-z^2}\operatorname{erfc}(-iz)$	*mad_num_wf()*
`dawson(z,rtol_,r_)`	$\frac{-i\sqrt{\pi}}{2}e^{-z^2}\operatorname{erf}(iz)$	*mad_num_dawson()*

1.5 Special Functions

The special function `fact()` supports negative integers as input as it uses extended factorial definition, and the values are cached to make its complexity in $O(1)$ after warmup.

The special function `rangle()` adjust the angle `a` versus the *previous* right angle `r`, e.g. during phase advance accumulation, to ensure proper value when passing through the $\pm 2k\pi$ boundaries.

Functions	Return values	C functions
`fact(n)`	$n!$	*mad_num_fact()*
`rangle(a,r)`	$a+2\pi\lfloor\frac{r-a}{2\pi}\rceil$	`round()`

1.6 Functions for Circular Sector

Basic functions for arc and cord lengths conversion rely on the following elementary relations:

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

where r stands for the radius and a for the angle of the Circular Sector.

Functions	Return values
`arc2cord(l,a)`	$l_{\text{arc}}\operatorname{sinc}\frac{a}{2}$
`arc2len(l,a)`	$l_{\text{arc}}\operatorname{sinc}\frac{a}{2}\cos a$
`cord2arc(l,a)`	$\frac{l_{\text{cord}}}{\operatorname{sinc}\frac{a}{2}}$
`cord2len(l,a)`	$l_{\text{cord}}\cos a$
`len2arc(l,a)`	$\frac{l}{\operatorname{sinc}\frac{a}{2}\,cosa}$
`len2cord(l,a)`	$\frac{l}{\cos a}$

2 Operators as Functions

The module `MAD.gfunc` provides many functions that are named version of operators and useful when operators cannot be used directly, e.g. when passed as argument or to compose together. These functions can also be retrieved from the module `MAD.gfunc.opstr` by their associated string (if available).

2.1 Math Operators

Functions for math operators are wrappers to associated mathematical operators, which themselves can be overridden by their associated metamethods.

Functions	Return values	Operator string	Metamethods
`unm(x)`	$-x$	`"~"`	`__unm(x,_)`
`inv(x)`	$1/x$	`"1/"`	`__div(1,x)`
`sqr(x)`	$x \cdot x$	`"^2"`	`__mul(x,x)`
`add(x,y)`	$x + y$	`"+"`	`__add(x,y)`
`sub(x,y)`	$x - y$	`"-"`	`__sub(x,y)`
`mul(x,y)`	$x \cdot y$	`"*"`	`__mul(x,y)`
`div(x,y)`	x/y	`"/"`	`__div(x,y)`
`mod(x,y)`	$x \mod y$	`"%"`	`__mod(x,y)`
`pow(x,y)`	x^y	`"^"`	`__pow(x,y)`

2.2 Element Operators

Functions for element-wise operators[4] are wrappers to associated mathematical operators of vector-like objects, which themselves can be overridden by their associated metamethods.

Functions	Return values	Operator string	Metamethods
`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_)`	$x .\hat{} y$	`".^"`	`__epow(x,y,r_)`

[4] Element-wise operators are not available directly in the programming language, here we use the Matlab-like notation for convenience.

2.3 Logical Operators

Functions for logical operators are wrappers to associated logical operators.

Functions	Return values	Operator string
`lfalse()`	`true`	`"T"`
`ltrue()`	`false`	`"F"`
`lnot(x)`	$\neg x$	`"!"`
`lbool(x)`	$\neg\neg x$	`"!!"`
`land(x,y)`	$x \wedge y$	`"&&"`
`lor(x,y)`	$x \vee y$	`"||"`
`lnum(x)`	$\neg x \to 0, \neg\neg x \to 1$	`"!#"`

2.4 Relational Operators

Functions for relational operators are wrappers to associated logical operators, which themselves can be overridden by their associated metamethods. Relational ordering operators are available only for objects that are ordered.

Functions	Return values	Operator string	Metamethods
`eq(x,y)`	$x = y$	`"=="`	`__eq(x,y)`
`ne(x,y)`	$x \neq y$	`"!="` or `"~="`	`__eq(x,y)`
`lt(x,y)`	$x < y$	`"<"`	`__lt(x,y)`
`le(x,y)`	$x \leq y$	`"<="`	`__le(x,y)`
`gt(x,y)`	$x > y$	`">"`	`__le(y,x)`
`ge(x,y)`	$x \geq y$	`">="`	`__lt(y,x)`
`cmp(x,y)`	$(x > y) - (x < y)$	`"?="`	

The special relational operator `cmp()` returns the number `1` for $x < y$, `-1` for $x > y$, and `0` otherwise.

2.5 Object Operators

Functions for object operators are wrappers to associated object operators, which themselves can be overridden by their associated metamethods.

Functions	Return values	Operator string	Metamethods
`get(x,k)`	$x[k]$	`"->"`	`__index(x,k)`
`set(x,k,v)`	$x[k] = v$	`"<-"`	`__newindex(x,k,v)`
`len(x)`	$\#x$	`"#"`	`__len(x)`
`cat(x,y)`	$x..y$	`".."`	`__concat(x,y)`
`call(x,...)`	$x(...)$	`"()"`	`__call(x,...)`

3 Bitwise Functions

Functions for bitwise operations are those from the LuaJIT module `bit` and imported into the module `MAD.gfunc` for convenience, see http://bitop.luajit.org/api.html for details. Note that all these functions have *value semantic* and normalise their arguments to the numeric range of a 32 bit integer before use.

Functions	Return values
`tobit(x)`	Return the normalized value of `x` to the range of a 32 bit integer
`tohex(x,n_)`	Return the hex string of `x` with `n` digits ($n < 0$ use caps)
`bnot(x)`	Return the bitwise reverse of `x` bits
`band(x,...)`	Return the bitwise *AND* of all arguments
`bor(x,...)`	Return the bitwise *OR* of all arguments
`bxor(x,...)`	Return the bitwise *XOR* of all arguments
`lshift(x,n)`	Return the bitwise left shift of `x` by `n` bits with 0-bit shift-in
`rshift(x,n)`	Return the bitwise right shift of `x` by `n` bits with 0-bit shift-in
`arshift(x,n)`	Return the bitwise right shift of `x` by `n` bits with sign bit shift-in
`rol(x,n)`	Return the bitwise left rotation of `x` by `n` bits
`ror(x,n)`	Return the bitwise right rotation of `x` by `n` bits
`bswap(x)`	Return the swapped bytes of `x`, i.e. convert big endian to/from little endian

3.1 Flags Functions

A flag is 32 bit unsigned integer used to store up to 32 binary states with the convention that `0` means disabled/cleared and 1 means enabled/set. Functions on flags are useful aliases to, or combination of, bitwise operations to manipulate their states (i.e. their bits). Flags are mainly used by the object model to keep track of hidden and user-defined states in a compact and efficient format.

Functions	Return values
`bset(x,n)`	Return the flag `x` with state `n` enabled
`bclr(x,n)`	Return the flag `x` with state `n` disabled
`btst(x,n)`	Return `true` if state `n` is enabled in `x`, `false` otherwise
`fbit(n)`	Return a flag with only state `n` enabled
`fnot(x)`	Return the flag `x` with all states flipped
`fset(x,...)`	Return the flag `x` with disabled states flipped if enabled in any flag passed as argument
`fcut(x,...)`	Return the flag `x` with enabled states flipped if disabled in any flag passed as argument
`fclr(x,f)`	Return the flag `x` with enabled states flipped if enabled in `f`
`ftst(x,f)`	Return `true` if all states enabled in `f` are enabled in `x`, `false` otherwise
`fall(x)`	Return `true` if all states are enabled in `x`, `false` otherwise
`fany(x)`	Return `true` if any state is enabled in `x`, `false` otherwise

4 Special Functions

The module `MAD.gfunc` provides some useful functions when passed as argument or composed with other functions.

Functions	Return values
`narg(...)`	Return the number of arguments
`ident(...)`	Return all arguments unchanged, i.e. functional identity
`fnil()`	Return `nil`, i.e. functional nil
`ftrue()`	Return `true`, i.e. functional true
`ffalse()`	Return `false`, i.e. functional false
`fzero()`	Return `0`, i.e. functional zero
`fone()`	Return `1`, i.e. functional one
`first(a)`	Return first argument and discard the others
`second(a,b)`	Return second argument and discard the others
`third(a,b,c)`	Return third argument and discard the others
`swap(a,b)`	Return first and second arguments swapped and discard the other arguments
`swapv(a,b,...)`	Return first and second arguments swapped followed by the other arguments
`echo(...)`	Return all arguments unchanged after echoing them on stdout

5 C API

These functions are provided for performance reason and compliance with the C API of other modules.

int **mad_num_sign**(*num_t* x)

Return an integer amongst `{-1, 0, 1}` representing the sign of the *number* `x`.

int **mad_num_sign1**(*num_t* x)

Return an integer amongst `{-1, 1}` representing the sign of the *number* `x`.

num_t **mad_num_fact**(int n)

Return the extended factorial the *number* `x`.

num_t **mad_num_powi**(*num_t* x, int n)

Return the *number* `x` raised to the power of the *integer* `n` using a fast algorithm.

num_t **mad_num_sinc**(*num_t* x)

Return the sine cardinal of the *number* `x`.

num_t **mad_num_sinhc**(*num_t* x)

Return the hyperbolic sine cardinal of the *number* `x`.

num_t **mad_num_asinc**(*num_t* x)

Return the arc sine cardinal of the *number* `x`.

num_t **mad_num_asinhc**(*num_t* x)

Return the hyperbolic arc sine cardinal of the *number* `x`.

num_t **mad_num_wf**(*num_t* x, *num_t* relerr)

Return the Faddeeva function of the *number* `x`.

num_t **mad_num_erf**(*num_t* x, *num_t* relerr)

Return the error function of the *number* `x`.

num_t **mad_num_erfc**(*num_t* x, *num_t* relerr)

Return the complementary error function of the *number* `x`.

num_t **mad_num_erfcx**(*num_t* x, *num_t* relerr)

Return the scaled complementary error function of the *number* `x`.

num_t **mad_num_erfi**(*num_t* x, *num_t* relerr)

Return the imaginary error function of the *number* `x`.

num_t **mad_num_dawson**(*num_t* x, *num_t* relerr)

Return the Dawson integral for the *number* `x`.

6 References

Chapter 31. Functors

This chapter describes how to create, combine and use *functors* from the `MAD` environment. Functors are objects that behave like functions with *callable* semantic, and also like readonly arrays with *indexable* semantic, where the index is translated as a unique argument into the function call. They are mainly used by the object model to distinguish them from functions which are interpreted as deferred expressions and evaluated automatically on reading, and by the Survey and Track tracking codes to handle (user-defined) actions.

1 Constructors

This module provides mostly constructors to create functors from functions, functors and any objects with *callable* semantic, and combine them all together.

`functor`(*f*)

Return a *functor* that encapsulates the function (or any callable object) `f`. Calling the returned functor is like calling `f` itself with the same arguments.

`compose`(*f*, *g*)

Return a *functor* that encapsulates the composition of `f` and `g`. Calling the returned functor is like calling $(f \circ g)(\dots)$. The operator `f ^ g` is a shortcut for *`compose`* if `f` is a *functor*.

`chain`(*f*, *g*)

Return a *functor* that encapsulates the calls chain of `f` and `g`. Calling the returned functor is like calling $f(\dots); g(\dots)$. The operator `f .. g` is a shortcut for *`chain`* if `f` is a *functor*.

`achain`(*f*, *g*)

Return a *functor* that encapsulates the *AND*-ed calls chain of `f` and `g`. Calling the returned functor is like calling $f(\dots) \wedge g(\dots)$.

`ochain`(*f*, *g*)

Return a *functor* that encapsulates the *OR*-ed calls chain of `f` and `g`. Calling the returned functor is like calling $f(\dots) \vee g(\dots)$.

`bind1st`(*f*, *a*)

Return a *functor* that encapsulates `f` and binds `a` as its first argument. Calling the returned functor is like calling $f(a, \dots)$.

`bind2nd`(*f*, *b*)

Return a *functor* that encapsulates `f` and binds `b` as its second argument. Calling the returned functor is like calling $f(a, b, \dots)$ where `a` may or may not be provided.

`bind3rd`(*f*, *c*)

Return a *functor* that encapsulates `f` and binds `c` as its third argument. Calling the returned functor is like calling $f(a, b, c, \dots)$ where `a` and `b` may or may not be provided.

`bind2st`(*f*, *a*, *b*)

Return a *functor* that encapsulates `f` and binds `a` and `b` as its two first arguments. Calling the returned functor is like calling $f(a, b, \dots)$.

bind3st(*f*, *a*, *b*, *c*)

Return a *functor* that encapsulates `f` and binds `a`, `b` and `c` as its three first arguments. Calling the returned functor is like calling $f(a, b, c, \dots)$.

bottom()

Return a *functor* that encapsulates the identity function `ident` to define the *bottom* symbol of functors. Bottom is also available in the operator strings table `opstr` as `"_|_"`.

2 Functions

is_functor(*a*)

Return `true` if `a` is a *functor*, `false` otherwise. This function is only available from the module `MAD.typeid`.

Chapter 32. Monomials

This chapter describes Monomial objects useful to encode the variables powers of Multivariate Taylor Series used by the Differential Algebra library of MAD-NG. The module for monomials is not exposed, only the contructor is visible from the `MAD` environment and thus, monomials must be handled directly by their methods. Monomial objects do not know to which variables the stored orders belong, the relationship is only through the indexes. Note that monomials are objects with reference semantic that store variable orders as 8-bit unsigned integers, thus arithmetic on variable orders occurs in the ring $\mathbb{N}/2^8\mathbb{N}$.

1 Constructors

The constructor for *monomial* is directly available from the `MAD` environment.

`monomial`(*[len_,] ord_*)

Return a *monomial* of size `len` with the variable orders set to the values given by `ord`, as computed by `mono:fill(ord_)`. If `ord` is omitted then `len` must be provided. Default: `len_ = #ord`, `ord_ = 0`.

2 Attributes

`mono.`**`n`**

The number of variable orders in `mono`, i.e. its size or length.

3 Functions

`is_monomial`(*a*)

Return `true` if `a` is a *monomial*, `false` otherwise. This function is only available from the module `MAD.typeid`.

4 Methods

The optional argument `r_` represents a destination placeholder for results.

`mono:`**`same`**(*n_*)

Return a monomial of length `n` filled with zeros. Default: `n_ = #mono`.

`mono:`**`copy`**(*r_*)

Return a copy of `mono`.

`mono:`**`fill`**(*ord_*)

Return `mono` with the variable orders set to the values given by `ord`. Default: `ord_ = 0`.

- If `ord` is a *number* then all variable orders are set to the value of `ord`.

- If `ord` is a *list* then all variable orders are set to the values given by `ord`.
- If `ord` is a *string* then all variable orders are set to the values given by `ord`, where each character in the set `[0-9A-Za-z]` is interpreted as a variable order in the Basis 62, e.g. the string `"Bc"` will be interpreted as a monomial with variable orders 11 and 38. Characters not in the set `[0-9A-Za-z]` are not allowed and lead to an undefined behavior, meaning that orders ≥ 62 cannot be safely specified through strings.

`mono:min()`

Return the minimum variable order of `mono`.

`mono:max()`

Return the maximum variable order of `mono`.

`mono:ord()`

Return the order of `mono`, that is the sum of all the variable orders.

`mono:ordp(`*step_*`)`

Return the product of the variable orders of `mono` at every `step`. Default: `step_ = 1`.

`mono:ordpf(`*step_*`)`

Return the product of the factorial of the variable orders of `mono` at every `step`. Default: `step_ = 1`.

`mono:add(`*mono2*, *r_*`)`

Return the sum of the monomials `mono` and `mono2`, that is the sum of the all their variable orders, i.e. $(o_1 + o_2) \mod 256$ where o_1 and o_2 are two variable orders at the same index in `mono` and `mono2`.

`mono:sub(`*mono2*, *r_*`)`

Return the difference of the monomials `mono` and `mono2`, that is the subtraction of the all their variable orders, i.e. $(o_1 - o_2) \mod 256$ where o_1 and o_2 are two variable orders at the same index in `mono` and `mono2`.

`mono:concat(`*mono2*, *r_*`)`

Return the concatenation of the monomials `mono` and `mono2`.

`mono:reverse(`*r_*`)`

Return the reverse of the monomial `mono`.

`mono:totable()`

Return a *list* containing all the variable orders of `mono`.

`mono:tostring(`*sep_*`)`

Return a *string* containing all the variable orders of `mono` encoded with characters in the set `[0-9A-Za-z]` and separated by the *string* `sep`. Default: `sep_ = ''`.

5 Operators

#mono

Return the number of variable orders in `mono`, i.e. its length.

mono[n]

Return the variable order at index `n` for `1 <= n <= #mono`, `nil` otherwise.

mono[n] = v

Assign the value `v` to the variable order at index `n` for `1 <= n <= #mono`, otherwise raise an *"out of bounds"* error.

mono + mono2

Equivalent to `mono:add(mono2)`.

mono - mono2

Equivalent to `mono:sub(mono2)`.

mono < mono2

Return `false` if one variable order in `mono` is greater or equal to the variable order at the same index in `mono2`, `true` otherwise.

mono <= mono2

Return `false` if one variable order in `mono` is greater than the variable order at the same index in `mono2`, `true` otherwise.

mono == mono2

Return `false` if one variable order in `mono` is not equal to the variable order at the same index in `mono2`, `true` otherwise.

mono .. mono2

Equivalent to `mono:concat(mono2)`.

6 Iterators

ipairs(*mono*)

Return an *ipairs* iterator suitable for generic `for` loops. The generated values are those returned by `mono[i]`.

7 C API

type **ord_t**

The variable order type, which is an alias for 8-bit unsigned integer. In the C API, monomials are arrays of variable orders with their size `n` tracked separately, i.e. `a[n]`.

ssz_t **mad_mono_str**(*ssz_t* n, *ord_t* a[*n*], *str_t* s)

Return the number of converted characters from the *string* `s` into variable orders stored to the monomial `a[n]`, as decribed in the method `:fill()`.

str_t **mad_mono_prt**(*ssz_t* n, const *ord_t* a[*n*], char s[*n* + 1])

Return the *string* `s` filled with characters resulting from the conversion of the variable orders given in the monomial `a[n]`, as decribed in the method `:tostring()`.

void **mad_mono_fill**(*ssz_t* n, *ord_t* a[*n*], *ord_t* v)

Fill the monomial `a[n]` with the variable order `v`.

void **mad_mono_copy**(*ssz_t* n, const *ord_t* a[*n*], *ord_t* r[*n*])

Copy the monomial `a[n]` to the monomial `r[n]`.

ord_t **mad_mono_min**(*ssz_t* n, const *ord_t* a[*n*])

Return the minimum variable order of the monomial `a[n]`.

ord_t **mad_mono_max**(*ssz_t* n, const *ord_t* a[*n*])

Return the minimum variable order of the monomial `a[n]`.

int **mad_mono_ord**(*ssz_t* n, const *ord_t* a[*n*])

Return the order of the monomial `a[n]`.

num_t **mad_mono_ordp**(*ssz_t* n, const *ord_t* a[*n*], *idx_t* stp)

Return the product of the variable orders of the monomial `a[n]` at every `stp`.

num_t **mad_mono_ordpf**(*ssz_t* n, const *ord_t* a[*n*], *idx_t* stp)

Return the product of the factorial of the variable orders of the monomial `a[n]` at every `stp`.

log_t **mad_mono_eq**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*])

Return `FALSE` if one variable order in monomial `a[n]` is not equal to the variable order at the same index in monomial `b[n]`, `TRUE` otherwise.

log_t **mad_mono_lt**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*])

Return `FALSE` if one variable order in monomial `a[n]` is greater or equal to the variable order at the same index in monomial `b[n]`, `TRUE` otherwise.

log_t **mad_mono_le**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*])

Return `FALSE` if one variable order in monomial `a[n]` is greater than the variable order at the same index in monomial `b[n]`, `TRUE` otherwise.

int **mad_mono_cmp**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*])

Return the difference between the first variable orders that are not equal for a given index starting from the beginning in monomials `a[n]` and `b[n]`.

int **mad_mono_rcmp**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*])

Return the difference between the first variable orders that are not equal for a given index starting from the end in monomials `a[n]` and `b[n]`.

void **mad_mono_add**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*], *ord_t* r[*n*])

Put the sum of the monomials `a[n]` and `b[n]` in the monomial `r[n]`.

void **mad_mono_sub**(*ssz_t* n, const *ord_t* a[*n*], const *ord_t* b[*n*], *ord_t* r[*n*])

Put the difference of the monomials `a[n]` and `b[n]` in the monomial `r[n]`.

void **mad_mono_cat**(*ssz_t* n, const *ord_t* a[*n*], *ssz_t* m, const *ord_t* b[*m*], *ord_t* r[*n* + *m*])

Put the concatenation of the monomials `a[n]` and `b[m]` in the monomial `r[n+m]`.

void **mad_mono_rev**(*ssz_t* n, const *ord_t* a[*n*], *ord_t* r[*n*])

Put the reverse of the monomial `a[n]` in the monomial `r[n]`.

void **mad_mono_print**(*ssz_t* n, const *ord_t* a[*n*], FILE *fp_)

Print the monomial `a[n]` to the file `fp`. Default: `fp_ = stdout`.

Chapter 33. Numerical Ranges

This chapter describes *range* and *logrange* objects that are useful abstaction of numerical loops, intervals, discrete sets, (log)lines and linear spaces. The module for numerical ranges is not exposed, only the constructors are visible from the `MAD` environment and thus, numerical ranges must be handled directly by their methods. Note that *range* and *logrange* have value semantic like *number*.

1 Constructors

The constructors for *range* and *logrange* are directly available from the `MAD` environment, except for the special case of the concatenation operator applied to two or three *number*, which is part of the language definition as a MAD-NG extension. The *logrange* behave as a the *range* but they work on logarithmic scale. All constructor functions adjust the value of `step` to ensure stable sizes and iterators across platforms (see the method `adjust` for details).

`start..`**`stop`**

`start..stop..`**`step`**

The concatenation operator applied to two or three numbers creates a *range* and does not perform any adjustment of `step`. The default step for the first form is one.

`range``(`*[start_,] stop, step_*`)`

Return a *range* object starting at `start`, ending at `stop` (included), with increments of size `step`. Default: `start_ = 1, step_ = 1`.

`nrange``(`*[start_,] stop, size_*`)`

Return a *range* object starting at `start`, ending at `stop` (included), with `size` increments. Default: `start_ = 1, size_ = 100`.

`logrange``(`*[start_,] stop, step_*`)`

Return a *logrange* object starting at `start`, ending at `stop` (included), with increments of size `step`. Default: `start_ = 1, step_ = 1`.

`nlogrange``(`*[start_,] stop, size_*`)`

Return a *logrange* object starting at `start`, ending at `stop` (included), with `size` increments. Default: `start_ = 1, size_ = 100`.

`torange``(`*str*`)`

Return a *range* decoded from the string `str` containing a literal numerical ranges of the form `"a..b"` or `"a..b..c"` where `a`, `b` and `c` are literal numbers.

1.1 Empty Ranges

Empty ranges of size zero can be created by fulfilling the constraints `start > stop` and `step > 0` or `start < stop` and `step < 0` in *range* constructor.

1.2 Singleton Ranges

Singleton ranges of size one can be created by fulfilling the constraints `step > stop-start` for `start < stop` and `step < stop-start` for `stop < start` in *range* constructor or `size == 1` in *nrange* constructor. In this latter case, `step` will be set to `step = huge * sign(stop-start)`.

1.3 Constant Ranges

Constant ranges of infinite size can be created by fulfilling the constraints `start == stop` and `step == 0` in *range* constructor or `size == inf` in *nrange* constructor. The user must satify the constraint `start == stop` in both constructors to show its intention.

2 Attributes

`rng.start`

`rng.logstart`

The component *start* of the *range* and the *logrange* on a linear scale.

`rng.stop`

`rng.logstop`

The component *stop* of the *range* and the *logrange* on a linear scale.

`rng.step`

`rng.logstep`

The component *step* of the *range* and the *logrange* on a linear scale, which may slighlty differ from the value provided to the constructors due to adjustment.

3 Functions

`is_range`(*a*)

`is_logrange`(*a*)

Return `true` if `a` is respectively a *range* or a *logrange*, `false` otherwise. These functions are only available from the module `MAD.typeid`.

`isa_range`(*a*)

Return `true` if `a` is a *range* or a *logrange* (i.e. is-a range), `false` otherwise. This function is only available from the module `MAD.typeid`.

4 Methods

Unless specified, the object `rng` that owns the methods represents either a *range* or a *logrange*.

`rng:`**`is_empty`**`()`

Return `false` if `rng` contains at least one value, `true` otherwise.

`rng:`**`same`**`()`

Return `rng` itself. This method is the identity for objects with value semantic.

`rng:`**`copy`**`()`

Return `rng` itself. This method is the identity for objects with value semantic.

`rng:`**`ranges`**`()`

Return the values of `start`, `stop` and `step`, fully characterising the range `rng`.

`rng:`**`size`**`()`

Return the number of values contained by the range `rng`, i.e. its size that is the number of steps plus one.

`rng:`**`value`**`(`*x*`)`

Return the interpolated value at `x`, i.e. interpreting the range `rng` as a (log)line with equation `start + x * step`.

`rng:`**`get`**`(`*x*`)`

Return `rng:value(x)` if the result is inside the range's bounds, `nil` otherwise.

`rng:`**`last`**`()`

Return the last value inside the bounds of the range `rng`, `nil` otherwise.

`rng:`**`adjust`**`()`

Return a range with a `step` adjusted.

The internal quantity `step` is adjusted if the computed size is close to an integer by $\pm 10^{-12}$. Then the following properties should hold even for rational binary numbers given a consistent input for `start`, `stop`, `step` and `size`:

- `#range(start, stop, step) == size`
- `nrange(start, stop, size):step() == step`
- `range (start, stop, step):value(size-1) == stop`

The maximum adjustment is `step = step * (1-eps)^2`, beyond this value it is the user reponsibility to provide better inputs.

`rng:`**`bounds`**`()`

Return the values of `start`, `last` (as computed by *`rng:last()`*) and `step` (made positive) characterising the boundaries of the range `rng`, i.e. interpreted as an interval, `nil` otherwise.

`rng:`**`overlap`**`(`*rng2*`)`

Return `true` if `rng` and `rng2` overlap, i.e. have intersecting bounds, `false` otherwise.

`rng:reverse()`

Return a range which is the reverse of the range `rng`, i.e. swap `start` and `stop`, and reverse `step`.

`rng:log()`

Return a *logrange* built by converting the *range* `rng` to logarithmic scale.

`rng:unm()`

Return a range with all components `start`, `stop` and `step` negated.

`rng:add(`*num*`)`

Return a range with `start` and `stop` shifted by `num`.

`rng:sub(`*num*`)`

Return a range with `start` and `stop` shifted by `-num`.

`rng:mul(`*num*`)`

Return a range with `stop` and `step` scaled by `num`.

`rng:div(`*num*`)`

Return a range with `stop` and `step` scaled by `1/num`.

`rng:tostring()`

Return a *string* encoding the range `rng` into a literal numerical ranges of the form `"a..b"` or `"a..b..c"` where `a`, `b` and `c` are literal numbers.

`rng:totable()`

Return a *table* filled with `#rng` values computed by *rng:value()*. Note that ranges are objects with a very small memory footprint while the generated tables can be huge.

5 Operators

`#rng`

Return the number of values contained by the range `rng`, i.e. it is equivalent to *rng:size()*.

`rng[n]`

Return the value at index `n` contained by the range `rng`, i.e. it is equivalent to `rng:get(round(n-1))`.

`-rng`

Equivalent to *rng:unm()*.

`rng + num`

`num + rng`

Equivalent to `rng:add(num)`.

`rng - num`

Equivalent to `rng:sub(num)`.

num - rng

Equivalent to `rng:unm():add(num)`.

num * rng

rng * num

Equivalent to `rng:mul(num)`.

rng / num

Equivalent to `rng:div(num)`.

rng == rng2

Return `true` if `rng` and `rng2` are of same king, have equal `start` and `stop`, and their `step` are within one `eps` from each other, `false` otherwise.

6 Iterators

ipairs(*rng*)

Return an *ipairs* iterator suitable for generic `for` loops. The generated values are those returned by `rng:value(i)`.

Chapter 34. Random Numbers

The module `gmath` provides few Pseudo-Random Number Generators (PRNGs).The defaut implementation is the *Xoshiro256*** (XOR/shift/rotate) variant of the XorShift PRNG familly [XORSHFT03], an all-purpose, rock-solid generator with a period of $2^{256} - 1$ that supports long jumps of period 2^{128}. This PRNG is also the default implementation of recent versions of Lua (not LuaJIT, see below) and GFortran. See https://prng.di.unimi.it for details about Xoshiro/Xoroshiro PRNGs.

The module `math` of LuaJIT provides an implementation of the *Tausworthe* PRNG [TAUSWTH96], which has a period of 2^{223} but doesn't support long jumps, and hence uses a single global PRNG.

The module `gmath` also provides an implementation of the simple global PRNG of MAD-X for comparison.

It's worth mentionning that none of these PRNG are cryptographically secure generators, they are nevertheless superior to the commonly used *Mersenne Twister* PRNG [MERTWIS98], with the exception of the MAD-X PRNG.

All PRNG *functions* (except constructors) are wrappers around PRNG *methods* with the same name, and expect an optional PRNG `prng_` as first parameter. If this optional PRNG `prng_` is omitted, i.e. not provided, these functions will use the current global PRNG by default.

1 Contructors

`randnew``()`

Return a new Xoshiro256** PRNG with a period of 2^{128} that is garuanteed to not overlapp with any other Xoshiro256** PRNGs, unless it is initialized with a seed.

`xrandnew``()`

Return a new MAD-X PRNG initialized with default seed 123456789. Hence, all new MAD-X PRNG will generate the same sequence until they are initialized with a user-defined seed.

2 Functions

`randset``(`*prng_*`)`

Set the current global PRNG to `prng` (if provided) and return the previous global PRNG.

`is_randgen``(`*a*`)`

Return `true` if `a` is a PRNG, `false` otherwise. This function is also available from the module `MAD.typeid`.

`is_xrandgen``(`*a*`)`

Return `true` if `a` is a MAD-X PRNG, `false` otherwise. This function is only available from the module `MAD.typeid`.

3 Methods

All methods are also provided as functions from the module `MAD.gmath` for convenience. If the PRNG is not provided, the current global PRNG is used instead.

`prng:`**`randseed`**`(`*seed*`)`

`randseed``(`*[prng_,] seed*`)`

Set the seed of the PRNG `prng` to `seed`.

`prng:`**`rand`**`()`

`rand``(`*prng_*`)`

Return a new pseudo-random number in the range `[0, 1)` from the PRNG `prng`.

`prng:`**`randi`**`()`

`randi``(`*prng_*`)`

Return a new pseudo-random number in the range of a *u64_t* from the PRNG `prng` (*u32_t* for the MAD-X PRNG), see C API below for details.

`prng:`**`randn`**`()`

`randn``(`*prng_*`)`

Return a new pseudo-random gaussian number in the range `[-inf, +inf]` from the PRNG `prng` by using the Box-Muller transformation (Marsaglia's polar form) to a peuso-random number in the range `[0, 1)`.

`prng:`**`randtn`**`(`*cut_*`)`

`randtn``(`*[prng_,] cut_*`)`

Return a new truncated pseudo-random gaussian number in the range `[-cut_, +cut_]` from the PRNG `prng` by using iteratively the method *prng:randn()*. This simple algorithm is actually used for compatibility with MAD-X. Default: `cut_ = +inf`.

`prng:`**`randp`**`(`*lmb_*`)`

`randp``(`*[prng_,] lmb_*`)`

Return a new pseudo-random poisson number in the range `[0, +inf]` from the PRNG `prng` with parameter $\lambda > 0$ by using the *inverse transform sampling* method on peuso-random numbers. Default: `lmb_ = 1`.

4 Iterators

`ipairs``(`*prng*`)`

Return an *ipairs* iterator suitable for generic `for` loops. The generated values are those returned by *prng:rand()*.

5 C API

type **prng_state_t**

type **xrng_state_t**

The Xoshiro256** and the MAD-X PRNG types.

num_t **mad_num_rand**(*prng_state_t**)

Return a pseudo-random double precision float in the range `[0, 1)`.

u64_t **mad_num_randi**(*prng_state_t**)

Return a pseudo-random 64 bit unsigned integer in the range `[0, ULLONG_MAX]`.

void **mad_num_randseed**(*prng_state_t**, *num_t* seed)

Set the seed of the PRNG.

void **mad_num_randjump**(*prng_state_t**)

Apply a jump to the PRNG as if 2^{128} pseudo-random numbers were generated. Hence PRNGs with different number of jumps will never overlap. This function is applied to new PRNGs with an incremental number of jumps.

num_t **mad_num_xrand**(*xrng_state_t**)

Return a pseudo-random double precision float in the range `[0, 1)` from the MAD-X PRNG.

u32_t **mad_num_xrandi**(*xrng_state_t**)

Return a pseudo-random 32 bit unsigned integer in the range `[0, UINT_MAX]` from the MAD-X PRNG.

void **mad_num_xrandseed**(*xrng_state_t**, u32_t seed)

Set the seed of the MAD-X PRNG.

6 References

Chapter 35. Complex Numbers

This chapter describes the *complex* numbers as supported by MAD-NG. The module for Complex numbers is not exposed, only the contructors are visible from the `MAD` environment and thus, complex numbers are handled directly by their methods or by the generic functions of the same name from the module `MAD.gmath`. Note that *complex* have value semantic like a pair of *number* equivalent to a C structure or an array `const num_t[2]` for direct compliance with the C API.

1 Types promotion

The operations on complex numbers may involve other data types like real numbers leading to few combinations of types. In order to simplify the descriptions, the generic names `num` and `cpx` are used for real and complex numbers respectively. The following table summarizes all valid combinations of types for binary operations involving at least one *complex* type:

Left Operand Type	Right Operand Type	Result Type
number	*complex*	*complex*
complex	*number*	*complex*
complex	*complex*	*complex*

2 Constructors

The constructors for *complex* numbers are directly available from the `MAD` environment, except for the special case of the imaginary postfix, which is part of the language definition.

`i`

The imaginary postfix that qualifies literal numbers as imaginary numbers, i.e. `1i` is the imaginary unit, and `1+2i` is the *complex* number $1 + 2i$.

`complex`(*re_*, *im_*)

Return the *complex* number equivalent to `re + im * 1i`. Default: `re_ = 0`, `im_ = 0`.

`tocomplex`(*str*)

Return the *complex* number decoded from the string `str` containing the literal complex number `"a+bi"` (with no spaces) where `a` and `b` are literal numbers, i.e. the strings `"1"`, `"2i"` and `"1+2i"` will give respectively the *complex* numbers $1 + 0i$, $0 + 2i$ and $1 + 2i$.

3 Attributes

`cpx.re`

The real part of the *complex* number `cpx`.

`cpx.im`

The imaginary part of the *complex* number `cpx`.

4 Functions

`is_complex`(*a*)

Return `true` if `a` is a *complex* number, `false` otherwise. This function is only available from the module `MAD.typeid`.

`is_scalar`(*a*)

Return `true` if `a` is a *number* or a *complex* number, `false` otherwise. This function is only available from the module `MAD.typeid`.

5 Methods

5.1 Operator-like Methods

Functions	Return values	Metamethods	C functions
`z:unm()`	$-z$	`__unm(z,_)`	
`z:add(z2)`	$z + z_2$	`__add(z,z2)`	
`z:sub(z2)`	$z - z_2$	`__sub(z,z2)`	
`z:mul(z2)`	$z \cdot z_2$	`__mul(z,z2)`	
`z:div(z2)`	z/z_2	`__div(z,z2)`	*mad_cpx_div_r()*[1]
`z:mod(z2)`	$z \mod z_2$	`__mod(z,z2)`	*mad_cpx_mod_r()*
`z:pow(z2)`	z^{z_2}	`__pow(z,z2)`	*mad_cpx_pow_r()*
`z:eq(z2)`	$z = z_2$	`__eq(z,z2)`	

[1] Division and inverse use a robust and fast complex division algorithm, see [CPXDIV] and [CPXDIV2] for details.

5.2 Real-like Methods

Functions	Return values	C functions
`z:abs()`	$\lvert z\rvert$	*mad_cpx_abs_r()*
`z:acos()`	$\cos^{-1} z$	*mad_cpx_acos_r()*
`z:acosh()`	$\cosh^{-1} z$	*mad_cpx_acosh_r()*
`z:acot()`	$\cot^{-1} z$	*mad_cpx_atan_r()*
`z:acoth()`	$\coth^{-1} z$	*mad_cpx_atanh_r()*
`z:asin()`	$\sin^{-1} z$	*mad_cpx_asin_r()*
`z:asinc()`	$\frac{\sin^{-1} z}{z}$	*mad_cpx_asinc_r()*
`z:asinh()`	$\sinh^{-1} x$	*mad_cpx_asinh_r()*
`z:asinhc()`	$\frac{\sinh^{-1} z}{z}$	*mad_cpx_asinhc_r()*
`z:atan()`	$\tan^{-1} z$	*mad_cpx_atan_r()*
`z:atanh()`	$\tanh^{-1} z$	*mad_cpx_atanh_r()*
`z:ceil()`	$\lceil \Re(z) \rceil + i \lceil \Im(z) \rceil$	
`z:cos()`	$\cos z$	*mad_cpx_cos_r()*
`z:cosh()`	$\cosh z$	*mad_cpx_cosh_r()*
`z:cot()`	$\cot z$	*mad_cpx_tan_r()*
`z:coth()`	$\coth z$	*mad_cpx_tanh_r()*
`z:exp()`	$\exp z$	*mad_cpx_exp_r()*
`z:floor()`	$\lfloor \Re(z) \rfloor + i \lfloor \Im(z) \rfloor$	
`z:frac()`	$z - \mathrm{trunc}(z)$	
`z:hypot(z2)`	$\sqrt{z^2 + z_2^2}$	[2]
`z:hypot3(z2,z3)`	$\sqrt{z^2 + z_2^2 + z_3^2}$	Page 207, 2
`z:inv(v_)`	$\frac{v}{z}$	*mad_cpx_inv_r()* Page 206, 1
`z:invsqrt(v_)`	$\frac{v}{\sqrt{z}}$	*mad_cpx_invsqrt_r()* Page 206, 1
`z:log()`	$\log z$	*mad_cpx_log_r()*
`z:log10()`	$\log_{10} z$	*mad_cpx_log10_r()*
`z:powi(n)`	z^n	*mad_cpx_powi_r()*
`z:round()`	$\lfloor \Re(z) \rceil + i \lfloor \Im(z) \rceil$	
`z:sin()`	$\sin z$	*mad_cpx_sin_r()*
`z:sinc()`	$\frac{\sin z}{z}$	*mad_cpx_sinc_r()*
`z:sinh()`	$\sinh z$	*mad_cpx_sinh_r()*
`z:sinhc()`	$\frac{\sinh z}{z}$	*mad_cpx_sinhc_r()*
`z:sqr()`	$z \cdot z$	
`z:sqrt()`	$\sqrt{z}$	*mad_cpx_sqrt_r()*
`z:tan()`	$\tan z$	*mad_cpx_tan_r()*
`z:tanh()`	$\tanh z$	*mad_cpx_tanh_r()*
`z:trunc()`	$\mathrm{trunc}\,\Re(z) + i\ \mathrm{trunc}\,\Im(z)$	
`z:unit()`	$\frac{z}{\lvert z\rvert}$	*mad_cpx_unit_r()*

In methods `inv()` and `invsqrt()`, default is `v_ = 1`.

[2] Hypot and hypot3 methods use a trivial implementation that may lead to numerical overflow/underflow.

5.3 Complex-like Methods

Functions	Return values	C functions
`z:cabs()`	$\|z\|$	*mad_cpx_abs_r()*
`z:carg()`	$\arg z$	*mad_cpx_arg_r()*
`z:conj()`	z^*	
`z:fabs()`	$\|\Re(z)\| + i\,\|\Im(z)\|$	
`z:imag()`	$\Im(z)$	
`z:polar()`	$\|z\|\, e^{i \arg z}$	*mad_cpx_polar_r()*
`z:proj()`	$\mathrm{proj}(z)$	*mad_cpx_proj_r()*
`z:real()`	$\Re(z)$	
`z:rect()`	$\Re(z)\cos\Im(z) + i\,\Re(z)\sin\Im(z)$	*mad_cpx_rect_r()*
`z:reim()`	$\Re(z), \Im(z)$	

5.4 Error-like Methods

Error-like methods call C wrappers to the corresponding functions from the Faddeeva library from the MIT, considered as one of the most accurate and fast implementation over the complex plane [FADDEEVA] (see `mad_num.c`).

Functions	Return values	C functions
`z:erf(rtol_)`	$\frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$	*mad_cpx_erf_r()*
`z:erfc(rtol_)`	$1 - \mathrm{erf}(z)$	*mad_cpx_erfc_r()*
`z:erfi(rtol_)`	$-i\,\mathrm{erf}(iz)$	*mad_cpx_erfi_r()*
`z:erfcx(rtol_)`	$e^{z^2}\,\mathrm{erfc}(z)$	*mad_cpx_erfcx_r()*
`z:wf(rtol_)`	$e^{-z^2}\,\mathrm{erfc}(-iz)$	*mad_cpx_wf_r()*
`z:dawson(rtol_)`	$\frac{-i\sqrt{\pi}}{2} e^{-z^2}\,\mathrm{erf}(iz)$	*mad_cpx_dawson_r()*

6 Operators

The operators on *complex* follow the conventional mathematical operations of Complex numbers.

`-cpx`

Return a *complex* resulting from the negation of the operand as computed by `cpx:unm()`.

`num + cpx`

`cpx + num`

`cpx + cpx`

Return a *complex* resulting from the sum of the left and right operands as computed by `cpx:add()`.

`num - cpx`

`cpx - num`

cpx - cpx

Return a *complex* resulting from the difference of the left and right operands as computed by `cpx:sub()`.

num * cpx

cpx * num

cpx * cpx

Return a *complex* resulting from the product of the left and right operands as computed by `cpx:mul()`.

num / cpx

cpx / num

cpx / cpx

Return a *complex* resulting from the division of the left and right operands as computed by `cpx:div()`. If the right operand is a complex number, the division uses a robut and fast algorithm implemented in *mad_cpx_div_r()*Page 206, 1.

num % cpx

cpx % num

cpx % cpx

Return a *complex* resulting from the rest of the division of the left and right operands, i.e. $x - y\lfloor\frac{x}{y}\rfloor$, as computed by `cpx:mod()`. If the right operand is a complex number, the division uses a robut and fast algorithm implemented in *mad_cpx_div_r()*Page 206, 1.

num ^ cpx

cpx ^ num

cpx ^ cpx

Return a *complex* resulting from the left operand raised to the power of the right operand as computed by `cpx:pow()`.

num == cpx

cpx == num

cpx == cpx

Return `false` if the real or the imaginary part differ between the left and right operands, `true` otherwise. A number `a` will be interpreted as $a + i0$ for the comparison.

7 C API

These functions are provided for performance reason and compliance (i.e. branch cut) with the C API of other modules dealing with complex numbers like the linear and the differential algebra. For the same reason, some functions hereafter refer to the section 7.3 of the C Programming Language Standard [ISOC99CPX].

num_t **mad_cpx_abs_r**(*num_t* x_re, *num_t* x_im)

Return the modulus of the *complex* `x` as computed by `cabs()`.

num_t **mad_cpx_arg_r**(*num_t* x_re, *num_t* x_im)

Return the argument in $[-\pi, +\pi]$ of the *complex* `x` as computed by `carg()`.

void **mad_cpx_unit_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the *complex* `x` divided by its modulus as computed by `cabs()`.

void **mad_cpx_proj_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the projection of the *complex* `x` on the Riemann sphere as computed by `cproj()`.

void **mad_cpx_rect_r**(*num_t* rho, *num_t* ang, *cpx_t* *r)

Put in `r` the rectangular form of the *complex* `rho * exp(i*ang)`.

void **mad_cpx_polar_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the polar form of the *complex* `x`.

void **mad_cpx_inv_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

cpx_t **mad_cpx_inv**(*cpx_t* x)

Put in `r` or return the inverse of the *complex* `x`.

void **mad_cpx_invsqrt_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the square root of the inverse of the *complex* `x`.

void **mad_cpx_div_r**(*num_t* x_re, *num_t* x_im, *num_t* y_re, *num_t* y_im, *cpx_t* *r)

cpx_t **mad_cpx_div**(*cpx_t* x, *cpx_t* y)

Put in `r` or return the *complex* `x` divided by the *complex* `y`.

void **mad_cpx_mod_r**(*num_t* x_re, *num_t* x_im, *num_t* y_re, *num_t* y_im, *cpx_t* *r)

Put in `r` the remainder of the division of the *complex* `x` by the *complex* `y`.

void **mad_cpx_pow_r**(*num_t* x_re, *num_t* x_im, *num_t* y_re, *num_t* y_im, *cpx_t* *r)

Put in `r` the *complex* `x` raised to the power of *complex* `y` using `cpow()`.

void **mad_cpx_powi_r**(*num_t* x_re, *num_t* x_im, int n, *cpx_t* *r)

cpx_t **mad_cpx_powi**(*cpx_t* x, int n)

Put in `r` or return the *complex* `x` raised to the power of the *integer* `n` using a fast algorithm.

void **mad_cpx_sqrt_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the square root of the *complex* `x` as computed by `csqrt()`.

void **mad_cpx_exp_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the exponential of the *complex* `x` as computed by `cexp()`.

void **mad_cpx_log_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the natural logarithm of the *complex* `x` as computed by `clog()`.

void **mad_cpx_log10_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the logarithm of the *complex* `x`.

void **mad_cpx_sin_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the sine of the *complex* `x` as computed by `csin()`.

void **mad_cpx_cos_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the cosine of the *complex* `x` as computed by `ccos()`.

void **mad_cpx_tan_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the tangent of the *complex* `x` as computed by `ctan()`.

void **mad_cpx_sinh_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the hyperbolic sine of the *complex* `x` as computed by `csinh()`.

void **mad_cpx_cosh_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the hyperbolic cosine of the *complex* `x` as computed by `ccosh()`.

void **mad_cpx_tanh_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the hyperbolic tangent of the *complex* `x` as computed by `ctanh()`.

void **mad_cpx_asin_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the arc sine of the *complex* `x` as computed by `casin()`.

void **mad_cpx_acos_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the arc cosine of the *complex* `x` as computed by `cacos()`.

void **mad_cpx_atan_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the arc tangent of the *complex* `x` as computed by `catan()`.

void **mad_cpx_asinh_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the hyperbolic arc sine of the *complex* `x` as computed by `casinh()`.

void **mad_cpx_acosh_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the hyperbolic arc cosine of the *complex* `x` as computed by `cacosh()`.

void **mad_cpx_atanh_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

Put in `r` the hyperbolic arc tangent of the *complex* `x` as computed by `catanh()`.

void **mad_cpx_sinc_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

cpx_t **mad_cpx_sinc**(*cpx_t* x)

Put in `r` or return the sine cardinal of the *complex* `x`.

void **mad_cpx_sinhc_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

cpx_t **mad_cpx_sinhc**(*cpx_t* x)

Put in `r` or return the hyperbolic sine cardinal of the *complex* `x`.

void **mad_cpx_asinc_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

cpx_t **mad_cpx_asinc**(*cpx_t* x)

Put in `r` or return the arc sine cardinal of the *complex* `x`.

void **mad_cpx_asinhc_r**(*num_t* x_re, *num_t* x_im, *cpx_t* *r)

cpx_t **mad_cpx_asinhc**(*cpx_t* x)

Put in `r` or return the hyperbolic arc sine cardinal of the *complex* `x`.

void **mad_cpx_wf_r**(*num_t* x_re, *num_t* x_im, *num_t* relerr, *cpx_t* *r)

cpx_t **mad_cpx_wf**(*cpx_t* x, *num_t* relerr)

Put in `r` or return the Faddeeva function of the *complex* `x`.

void **mad_cpx_erf_r**(*num_t* x_re, *num_t* x_im, *num_t* relerr, *cpx_t* *r)

cpx_t **mad_cpx_erf**(*cpx_t* x, *num_t* relerr)

Put in `r` or return the error function of the *complex* `x`.

void **mad_cpx_erfc_r**(*num_t* x_re, *num_t* x_im, *num_t* relerr, *cpx_t* *r)

cpx_t **mad_cpx_erfc**(*cpx_t* x, *num_t* relerr)

Put in `r` or return the complementary error function of the *complex* `x`.

void **mad_cpx_erfcx_r**(*num_t* x_re, *num_t* x_im, *num_t* relerr, *cpx_t* *r)

cpx_t **mad_cpx_erfcx**(*cpx_t* x, *num_t* relerr)

Put in `r` or return the scaled complementary error function of the *complex* `x`.

void **mad_cpx_erfi_r**(*num_t* x_re, *num_t* x_im, *num_t* relerr, *cpx_t* *r)

cpx_t **mad_cpx_erfi**(*cpx_t* x, *num_t* relerr)

Put in `r` or return the imaginary error function of the *complex* `x`.

void **mad_cpx_dawson_r**(*num_t* x_re, *num_t* x_im, *num_t* relerr, *cpx_t* *r)

cpx_t **mad_cpx_dawson**(*cpx_t* x, *num_t* relerr)

Put in `r` or return the Dawson integral for the *complex* `x`.

8 References

Chapter 36. Linear Algebra

This chapter describes the real *matrix*, complex *cmatrix* and integer *imatrix* objects as supported by MADNG. The module for Vector and Matrix is not exposed, only the contructors are visible from the `MAD` environment and thus, matrices are handled directly by their methods or by the generic functions of the same name from the module `MAD.gmath`. The *imatrix*, i.e. matrix of integers, are mainly used for indexing other types of matrix and therefore supports only a limited subset of the features. Column and row vectors are shortcuts for $[n \times 1]$ and $[1 \times n]$ matrices respectively. Note that *matrix*, *cmatrix* and *imatrix* are all defined as C structures containing their elements in row-major order for direct compliance with the C API.

1 Types promotion

The matrix operations may involve other data types like real and complex numbers leading to many combinations of types. In order to simplify the descriptions, the generic names `num`, `cpx` and `idx` (indexes) are used for real, complex and integer numbers respectively, `vec`, `cvec` and `ivec` for real, complex and integer vectors respectively, and `mat`, `cmat` and `imat` for real, complex and integer matrices respectively. For example, the sum of a complex number `cpx` and a real matrix `mat` gives a complex matrix `cmat`. The case of `idx` means that a *number* will be interpreted as an index and automatically rounded if it does not hold an integer value. The following table summarizes all valid combinations of types for binary operations involving at least one matrix type:

Left Operand Type	Right Operand Type	Result Type
number	*imatrix*	*imatrix*
imatrix	*number*	*imatrix*
imatrix	*imatrix*	*imatrix*
number	*matrix*	*matrix*
matrix	*number*	*matrix*
matrix	*matrix*	*matrix*
number	*cmatrix*	*cmatrix*
complex	*matrix*	*cmatrix*
complex	*cmatrix*	*cmatrix*
matrix	*complex*	*cmatrix*
matrix	*cmatrix*	*cmatrix*
cmatrix	*number*	*cmatrix*
cmatrix	*complex*	*cmatrix*
cmatrix	*matrix*	*cmatrix*
cmatrix	*cmatrix*	*cmatrix*

2 Constructors

The constructors for vectors and matrices are directly available from the `MAD` environment. Note that real, complex or integer matrix with zero size are not allowed, i.e. the smallest allowed matrix has sizes of $[1 \times 1]$.

vector(*nrow*)

cvector(*nrow*)

ivector(*nrow*)

Return a real, complex or integer column vector (i.e. a matrix of size $[n_{\text{row}} \times 1]$) filled with zeros. If `nrow` is a table, it is equivalent to `vector(#nrow):fill(nrow)`.

matrix(*nrow*, *ncol_*)

cmatrix(*nrow*, *ncol_*)

imatrix(*nrow*, *ncol_*)

Return a real, complex or integer matrix of size $[n_{\text{row}} \times n_{\text{col}}]$ filled with zeros. If `nrow` is a table, it is equivalent to `matrix(#nrow, #nrow[1] or 1):fill(nrow)`, and ignoring `ncol`. Default: `ncol_ = rnow`.

linspace(*[start_,] stop*, *size_*)

Return a real or complex column vector of length `size` filled with values equally spaced between `start` and `stop` on a linear scale. Note that numerical *range* can generate the same *real* sequence of values in a more compact form. Default: `start_ = 0`, `size_ = 100`.

logspace(*[start_,] stop*, *size_*)

Return a real or complex column vector of length `size` filled with values equally spaced between `start` and `stop` on a logarithmic scale. Note that numerical *logrange* can generate the same *real* sequence of values in a more compact form. Default: `start_ = 1`, `size_ = 100`.

3 Attributes

mat.**nrow**

The number of rows of the real, complex or integer matrix `mat`.

mat.**ncol**

The number of columns of the real, complex or integer matrix `mat`.

4 Functions

is_vector(*a*)

is_cvector(*a*)

is_ivector(*a*)

Return `true` if `a` is respectively a real, complex or integer matrix of size $[n_{\text{row}} \times 1]$ or $[1 \times n_{\text{row}}]$, `false` otherwise. These functions are only available from the module `MAD.typeid`.

isa_vector(*a*)

Return `true` if `a` is a real or complex vector (i.e. is-a vector), `false` otherwise. This function is only available from the module `MAD.typeid`.

isy_vector(*a*)

Return `true` if `a` is a real, complex or integer vector (i.e. is-any vector), `false` otherwise. This function is only available from the module `MAD.typeid`.

is_matrix(*a*)

is_cmatrix(*a*)

is_imatrix(*a*)

Return `true` if `a` is respectively a real, complex or integer matrix, `false` otherwise. These functions are only available from the module `MAD.typeid`.

isa_matrix(*a*)

Return `true` if `a` is a real or complex matrix (i.e. is-a matrix), `false` otherwise. This function is only available from the module `MAD.typeid`.

isy_matrix(*a*)

Return `true` if `a` is a real, complex or integer matrix (i.e. is-any matrix), `false` otherwise. This function is only available from the module `MAD.typeid`.

5 Methods

5.1 Special Constructors

`mat:`**vec**`()`

Return a vector of the same type as `mat` filled with the values of the elements of the vectorized real, complex or integer matrix `mat` equivalent to `mat:t():reshape(#mat,1)`.

`mat:`**vech**`()`

Return a vector of the same type as `mat` filled with the values of the elements of the half vectorized real, complex or integer *symmetric* matrix `mat`. The symmetric property can be pre-checked by the user with *mat:is_symm()*.

`mat:`**diag**(*k_*)

If `mat` is a matrix, return a column vector containing its k-th diagonal equivalent to `mat:getdiag(k)`. If `mat` is a vector, return a square matrix with its k-th diagonal set to the values of the elements of `mat` equivalent to `mat:same(n,n):setdiag(mat,k)` where `n = #mat+abs(k)`. Default: `k_ = 0`.

5.2 Sizes and Indexing

`mat:`**`size`**`()`

Return the number of elements `nrow * ncol` of the real, complex or integer matrix `mat` equivalent to `#mat`.

`mat:`**`bytesize`**`()`

Return the number of *bytes* used by the data storage of the real, complex or integer matrix `mat` equivalent to `#mat * sizeof(mat[1])`.

`mat:`**`sizes`**`()`

Return the number of rows `nrow` and columns `ncol` of the real, complex or integer matrix `mat`. Note that `#mat` returns the full size `nrow * ncol` of the matrix.

`mat:`**`tsizes`**`()`

Return the number of columns `ncol` and rows `nrow` (i.e. transposed sizes) of the real, complex or integer matrix `mat` equivalent to `swap(mat:sizes())`.

`mat:`**`getij`**`(`*ij_*, *ri_*, *rj_*`)`

Return two *ivector* or `ri` and `rj` containing the indexes `(i,j)` extracted from the *iterable* `ij` for the real, complex or integer matrix `mat`. If `ij` is a number, the two returned items are also numbers. This method is the reverse method of *`mat:getidx()`* to convert 1D indexes into 2D indexes for the given matrix sizes. Default: `ij_ = 1..#mat`.

`mat:`**`getidx`**`(`*ir_*, *jc_*, *rij_*`)`

Return an *ivector* or `rij` containing `#ir * #jc` vector indexes in row-major order given by the *iterable* `ir` and `jc` of the real, complex or integer matrix `mat`, followed by `ir` and `jc` potentially set from defaults. If both `ir` and `jc` are numbers, it returns a single number. This method is the reverse method of *`mat:getij()`* to convert 2D indexes into 1D indexes for the given matrix sizes. Default: `ir_ = 1..nrow`, `jc_ = 1..ncol`.

`mat:`**`getdidx`**`(`*k_*`)`

Return an *iterable* describing the indexes of the k-th diagonal of the real, complex or integer matrix `mat` where `-nrow <= k <= ncol`. This method is useful to build the 1D indexes of the k-th diagonal for the given matrix sizes. Default `k_ = 0`

5.3 Getters and Setters

`mat:`**`get`**`(`*i*, *j*`)`

Return the value of the element at the indexes `(i,j)` of the real, complex or integer matrix `mat` for `1 <= i <= nrow` and `1 <= j <= ncol`, `nil` otherwise.

`mat:`**`set`**`(`*i*, *j*, *v*`)`

Assign the value `v` to the element at the indexes `(i,j)` of the real, complex or integer matrix `mat` for `1 <= i <= nrow` and `1 <= j <= ncol` and return the matrix, otherwise raise an *"index out of bounds"* error.

`mat:`**`geti`**(*n*)

Return the value of the element at the vector index `n` of the real, complex or integer matrix `mat` for `1 <= n <= #mat`, i.e. interpreting the matrix as a vector, `nil` otherwise.

`mat:`**`seti`**(*n*, *v*)

Assign the value `v` to the element at the vector index `n` of the real, complex or integer matrix `mat` for `1 <= n <= #mat` and return the matrix, i.e. interpreting the matrix as a vector, otherwise raise an *"index out of bounds"* error.

`mat:`**`getvec`**(*ij*, *r_*)

Return a column vector or `r` containing the values of the elements at the vector indexes given by the *iterable* `ij` of the real, complex or integer matrix `mat`, i.e. interpreting the matrix as a vector.

`mat:`**`setvec`**(*ij*, *a*, *p_*, *s_*)

Return the real, complex or integer matrix `mat` after filling the elements at the vector indexes given by the *iterable* `ij`, i.e. interpreting the matrix as a vector, with the values given by `a` depending of its kind:

- if `a` is a *scalar*, it is will be used repetitively.
- if `a` is an *iterable* then the matrix will be filled with values from `a[n]` for `1 <= n <= #a` and recycled repetitively if `#a < #ij`.
- if `a` is a *callable*, then `a` is considered as a *stateless iterator*, and the matrix will be filled with the values `v` returned by iterating `s, v = a(p, s)`.

`mat:`**`insvec`**(*ij*, *a*)

Return the real, complex or integer matrix `mat` after inserting the elements at the vector indexes given by the *iterable* `ij`, i.e. interpreting the matrix as a vector, with the values given by `a` depending of its kind:

- if `a` is a *scalar*, it is will be used repetitively.
- if `a` is an *iterable* then the matrix will be filled with values from `a[n]` for `1 <= n <= #a`.

The elements after the inserted indexes are shifted toward the end of the matrix in row-major order and discarded if they go beyond the last index.

`mat:`**`remvec`**(*ij*)

Return the real, complex or integer matrix `mat` after removing the elements at the vector indexes given by the *iterable* `ij`, i.e. interpreting the matrix as a shrinking vector, and reshaped as a *column vector* of size `#mat - #ij`.

`mat:`**`swpvec`**(*ij*, *ij2*)

Return the real, complex or integer matrix `mat` after swapping the values of the elements at the vector indexes given by the *iterable* `ij` and `ij2`, i.e. interpreting the matrix as a vector.

`mat:`**`getsub`**(*ir_*, *jc_*, *r_*)

Return a $\lceil$ `#ir` $\times$ `#jc` $\rceil$ matrix or `r` containing the values of the elements at the indexes given by the *iterable* `ir` and `jc` of the real, complex or integer matrix `mat`. If `ir = nil`, `jc ~= nil` and `r` is a 1D *iterable*, then the latter is filled using column-major indexes. Default: as *`mat:getidx()`*.

`mat:`**`setsub`**(*ir_*, *jc_*, *a*, *p_*, *s_*)

Return the real, complex or integer matrix `mat` after filling the elements at the indexes given by the *iterable* `ir` and `jc` with the values given by `a` depending of its kind:

- if `a` is a *scalar*, it is will be used repetitively.
- if `a` is an *iterable* then the rows and columns will be filled with values from `a[n]` for `1 <= n <= #a` and recycled repetitively if `#a < #ir * #ic`.
- if `a` is a *callable*, then `a` is considered as a *stateless iterator*, and the columns will be filled with the values `v` returned by iterating `s, v = a(p, s)`.

If `ir = nil`, `jc ~= nil` and `a` is a 1D *iterable*, then the latter is used to filled the matrix in the column-major order. Default: as *mat:getidx()*.

`mat:`**`inssub`**(*ir_*, *jc_*, *a*)

Return the real, complex or integer matrix `mat` after inserting elements at the indexes `(i,j)` given by the *iterable* `ir` and `jc` with the values given by `a` depending of its kind:

- if `a` is a *scalar*, it is will be used repetitively.
- if `a` is an *iterable* then the rows and columns will be filled with values from `a[n]` for `1 <= n <= #a` and recycled repetitively if `#a < #ir * #ic`.

The values after the inserted indexes are pushed toward the end of the matrix, i.e. interpreting the matrix as a vector, and discarded if they go beyond the last index. If `ir = nil`, `jc ~= nil` and `a` is a 1D *iterable*, then the latter is used to filled the matrix in the column-major order. Default: as *mat:getidx()*.

`mat:`**`remsub`**(*ir_*, *jc_*)

Return the real, complex or integer matrix `mat` after removing the rows and columns at the indexes given by the *iterable* `ir` and `jc` and reshaping the matrix accordingly. Default: as *mat:getidx()*.

`mat:`**`swpsub`**(*ir_*, *jc_*, *ir2_*, *jc2_*)

Return the real, complex or integer matrix `mat` after swapping the elements at indexes given by the iterable *iterable* `ir` and `jc` with the elements at indexes given by *iterable* `ir2` and `jc2`. Default: as *mat:getidx()*.

`mat:`**`getrow`**(*ir*, *r_*)

Equivalent to *mat:getsub()* with `jc = nil`.

`mat:`**`setrow`**(*ir*, *a*, *p_*, *s_*)

Equivalent to *mat:setsub()* with `jc = nil`.

`mat:`**`insrow`**(*ir*, *a*)

Equivalent to *mat:inssub()* with `jc = nil`.

`mat:`**`remrow`**(*ir*)

Equivalent to *mat:remsub()* with `jc = nil`.

`mat:`**`swprow`**(*ir*, *ir2*)

Equivalent to *mat:swpsub()* with `jc = nil` and `jc2 = nil`.

`mat:`**`getcol`**(*jc*, *r_*)

Equivalent to *mat:getsub()* with `ir = nil`.

`mat:`**`setcol`**(*jc*, *a*, *p_*, *s_*)

Equivalent to *mat:setsub()* with `ir = nil`.

`mat:`**`inscol`**(*jc*, *a*)

Equivalent to *mat:inssub()* with `ir = nil`. If `a` is a matrix with `ncol > 1` then `a = 0` and it is followed by *mat:setsub()* with `ir = nil` to obtain the expected result.

`mat:`**`remcol`**(*jc*)

Equivalent to *mat:remsub()* with `ir = nil`.

`mat:`**`swpcol`**(*jc*, *jc2*)

Equivalent to *mat:swpsub()* with `ir = nil` and `ir2 = nil`.

`mat:`**`getdiag`**(*[k_,] r_*)

Return a column vector of length `min(nrow, ncol)-abs(k)` or `r` containing the values of the elements on the k-th diagonal of the real, complex or integer matrix `mat` using *mat:getvec()*. Default: as *mat:getdidx()*.

`mat:`**`setdiag`**(*a*, *[k_,] p_*, *s_*)

Return the real, complex or integer matrix `mat` after filling the elements on its k-th diagonal with the values given by `a` using *mat:setvec()*. Default: as *mat:getdidx()*.

5.4 Cloning and Reshaping

`mat:`**`copy`**(*r_*)

Return a matrix or `r` filled with a copy of the real, complex or integer matrix `mat`.

`mat:`**`same`**(*[nr_, nc_,] v_*)

Return a matrix with elements of the type of `v` and with `nr` rows and `nc` columns. Default: `v_ = mat[1]`, `nr_ = nrow`, `nc_ = ncol`.

`mat:`**`reshape`**(*nr_*, *nc_*)

Return the real, complex or integer matrix `mat` reshaped to the new sizes `nr` and `nc` that must result into an equal or smaller number of elements, or it will raise an *invalid new sizes* error. Default: `nr_ = #mat`, `nc_ = 1`.

`mat:`**`_reshapeto`**(*nr*, *nc_*)

Same as *mat:reshape()* except that `nr` must be explicitly provided as this method allows for a larger size than `mat` current size. A typical use of this method is to expand a vector after an explicit shrinkage, while keeping track of its original size, e.g. similar to `vector(100) :reshape(1):seti(1,1) :_reshapeto(2):seti(2,1)` that would raise an *"index out of bounds"* error without the call to `_reshapeto()`. Default `nc_ = 1`.

WARNING: This method is unsafe and may crash MAD-NG with, e.g. a Segmentation fault *, if wrongly used. It is the responsibility of the user to ensure that* `mat` *was created with a size greater than or equal to the new size.*

5.5 Matrix Properties

`mat:`**`is_const`**`(`*tol_*`)`

Return true if all elements are equal within the tolerance `tol`, false otherwise. Default: `tol_ = 0`.

`mat:`**`is_real`**`(`*tol_*`)`

Return true if the imaginary part of all elements are equal to zero within the tolerance `tol`, false otherwise. Default: `tol_ = 0`.

`mat:`**`is_imag`**`(`*tol_*`)`

Return true if the real part of all elements are equal to zero within the tolerance `tol`, false otherwise. Default: `tol_ = 0`.

`mat:`**`is_diag`**`(`*tol_*`)`

Return true if all elements off the diagonal are equal to zero within the tolerance `tol`, false otherwise. Default: `tol_ = 0`.

`mat:`**`is_symm`**`(`*[tol_,] [sk_,] c_*`)`

Return true if `mat` is a symmetric matrix, i.e. $M = M^*$ within the tolerance `tol`, false otherwise. It checks for a skew-symmetric matrix if `sk = true`, and for a Hermitian matrix if `c ~= false`, and a skew-Hermitian matrix if both are combined. Default: `tol_ = 0`.

`mat:`**`is_symp`**`(`*tol_*`)`

Return true if `mat` is a symplectic matrix, i.e. $M^* S_{2n} M = S_{2n}$ within the tolerance `tol`, false otherwise. Default: `tol_ = eps`.

5.6 Filling and Moving

`mat:`**`zeros`**`()`

Return the real, complex or integer matrix `mat` filled with zeros.

`mat:`**`ones`**`(`*v_*`)`

Return the real, complex or integer matrix `mat` filled with the value of `v`. Default: `v_ = 1`.

`mat:`**`eye`**`(`*v_*`)`

Return the real, complex or integer matrix `mat` filled with the value of `v` on the diagonal and zeros elsewhere. The name of this method comes from the spelling of the Identity matrix I. Default: `v_ = 1`.

`mat:`**`seq`**`(`*[v_,] d_*`)`

Return the real, complex or integer matrix `mat` filled with the indexes of the elements (i.e. starting at 1) and shifted by the value of `v`. The matrix is filled in the row-major order unless `d = 'col'`. Default: `v_ = 0`.

`mat:`**`random`**`(`*f_, ...*`)`

Return the real, complex or integer matrix `mat` filled with random values generated by `f(...)`, and called twice for each element for a *cmatrix*. Default: `f_ = math.random`.

`mat:`**`shuffle`**`()`

Return the real, complex or integer matrix `mat` with its elements randomly swapped using the Fisher–Yates or Knuth shuffle algorithm and `math.random` as the PRNG.

`mat:`**`symp`**`()`

Return the real or complex matrix `mat` filled with the block diagonal unitary Symplectic matrix sometimes named J_{2n} or S_{2n}. The matrix `mat` must be square with even number of rows and columns otherwise a *"2n square matrix expected"* error is raised.

`mat:`**`circ`**`(`*v*`)`

Return the real or complex matrix `mat` filled as a Circulant matrix using the values from the *iterable* `v`, and rotating elements for each row or column depending on the shape of `v`.

`mat:`**`fill`**`(`*a*`, `*p_*`, `*s_*`)`

Return the real, complex or integer matrix `mat` filled with values provided by `a` depending of its kind:

- if `a` is a *scalar*, it is equivalent to `mat:ones(a)`.
- if `a` is a *callable*, then:
  - if `p` and `s` are provided, then `a` is considered as a *stateless iterator*, and the matrix will be filled with the values `v` returned by iterating `s, v = a(p, s)`.
  - otherwise `a` is considered as a *generator*, and the matrix will be filled with values returned by calling `a(mat:get(i,j), i, j)`.
- if `a` is an *iterable* then:
  - if `a[1]` is also an *iterable*, the matrix will be filled with the values from `a[i][j]` for `1 <= i <= nrow` and `1 <= j <= ncol`, i.e. treated as a 2D container.
  - otherwise the matrix will be filled with values from `a[n]` for `1 <= n <= #mat`, i.e. treated as a 1D container.

`mat:`**`rev`**`(`*d_*`)`

Reverse the elements of the matrix `mat` according to the direction `d`:

- If `d = 'vec'`, it reverses the entire matrix.
- If `d = 'row'`, it reverses each row.
- If `d = 'col'`, it reverses each column.
- If `d = 'diag'`, it reverse the only the diagonal.

Default: `d_ = 'vec'`.

`mat:`**`roll`**`(`*nr_*`, `*nc_*`)`

Return the real, complex or integer matrix `mat` after rolling its rows by `nr` $\in \mathbb{Z}$ and then columns by `nc` $\in \mathbb{Z}$. Default: `nr_ = 0`, `nc_ = 0`.

`mat:`**`movev`**`(`*i*`, `*j*`, `*k*`, `*r_*`)`

Return the real, complex or integer matrix `r` after moving the elements in `mat[i..j]` to `r[k..k+j-i]` with `1 <= i <= j <= #mat` and `1 <= k <= k+j-i <= #r`. Default: `r_ = mat`.

`mat:`**`shiftv`**`(`*i*`, `*n_*`)`

Return the real, complex or integer matrix `mat` after shifting the elements in `mat[i..]` to `mat[i+n..]` if `n > 0` and in the opposite direction if `n < 0`, i.e. it is equivalent to `mat:movev(i, #mat-n, i+n)` for `n > 0` and to `mat:movev(i, #mat, i+n)` for `n < 0`. Default: `n_ = 1`.

5.7 Mapping and Folding

This section lists the high-order functions map, fold and their variants useful in functional programming[1], followed by sections that list their direct application.

`mat:`**`foreach`**`(`*`[ij_, ] f`*`)`

Return the real, complex or integer matrix `mat` after applying the *callable* `f` to the elements at the indexes given by the *iterable* `ij` using `f(mat[n], n)`, i.e. interpreting the matrix as a vector. Default: `ij_ = 1..#mat`.

`mat:`**`filter`**`(`*`[ij_, ] p, r_`*`)`

Return a matrix or `r` filled with the values of the elements of the real, complex or integer matrix `mat` at the indexes given by the *iterable* `ij` if they are selected by the *callable* predicate `p` using `p(mat[n], n) = true`, i.e. interpreting the matrix as a vector. This method returns next to the matrix, a *table* if `r` is a table or a *ivector* otherwise, containing the indexes of the selected elements returned. Default: `ij_ = 1..#mat`.

`mat:`**`filter_out`**`(`*`[ij_, ] p, r_`*`)`

Equivalent to `map:filter(ij_, compose(lnot,p), r_)`, where the functions *`compose()`* and `lnot()` are provided by the module `MAD.gfunc`.

`mat:`**`map`**`(`*`[ij_, ] f, r_`*`)`

Return a matrix or `r` filled with the values returned by the *callable* (or the operator string) `f` applied to the elements of the real, complex or integer matrix `mat` at the indexes given by the *iterable* `ij` using `f(mat[n], n)`, i.e. interpreting the matrix as a vector. If `r = 'in'` or `r = nil` and `ij ~= nil` then it is assigned `mat`, i.e. map in place. If `r = nil` still, then the type of the returned matrix is determined by the type of the value returned by `f()` called once before mapping. Default: `ij_ = 1..#mat`.

`mat:`**`map2`**`(`*`y, [ij_, ] f, r_`*`)`

Equivalent to *`mat:map()`* but with two arguments passed to `f`, i.e. using `f(mat[n], y[n], n)`.

`mat:`**`map3`**`(`*`y, z, [ij_, ] f, r_`*`)`

Equivalent to *`mat:map()`* but with three arguments passed to `f`, i.e. using `f(mat[n], y[n], z[n], n)`. Note that `f` cannot be an operator string, as only unary and binary operators are avalaible in such form.

`mat:`**`foldl`**`(`*`f, [x0_, ] [d_, ] r_`*`)`

Return a scalar, a vector or `r` filled with the values returned by the *callable* (or the operator string) `f` applied iteratively to the elements of the real, complex or integer matrix `mat` using the folding left (forward with increasing indexes) expression `v = f(v, mat[n])` starting at `x0` and running in the direction depending on the *string* `d`:

- If `d = 'vec'`, the folding left iteration runs on the entire matrix `mat` interpreted as a vector and a scalar is returned.
- If `d = 'row'`, the folding left iteration runs on the rows of the matrix `mat` and a column vector is returned.

[1] For *true* Functional Programming, see the module `MAD.lfun`, a binding of the LuaFun library adapted to the ecosystem of MAD-NG.

- If `d = 'col'`, the folding left iteration runs on the columns of the matrix `mat` and a row vector is returned.
- If `d = 'diag'`, the folding left iteration runs on the diagonal of the matrix `mat` and a scalar is returned.

Note that ommitting both `x0` and `d` implies to not specify `r` as well, otherwise the latter will be interpreted as `x0`. If `r = nil` and `d = 'row'` or `d = 'col'`, then the type of the returned vector is determined by the type of the value returned by `f()` called once before folding. Default: `x0 = mat[1]` (or first row or column element), `d = 'vec'`.

`mat:`**`foldr`**(*f*, *[x0_,] [d_,] r_*)

Same as *mat:foldl()* but the *callable* (or the operator string) `f` is applied iteratively using the folding right (backward with decreasing indexes) expression `v = f(mat[n], v)`. Default: `x0 = mat[#mat]` (or last row or column element), `d = 'vec'`.

`mat:`**`scanl`**(*f*, *[x0_,] [d_,] r_*)

Return a vector, a matrix or `r` filled with the values returned by the *callable* (or the operator string) `f` applied iteratively to the elements of the real, complex or integer matrix `mat` using the scanning left (forward with increasing indexes) expression `v = f(v, mat[n])` starting at `x0` and running in the direction depending on the *string* `d`:

- If `d = 'vec'`, the scanning left iteration runs on the entire matrix `mat` interpreted as a vector and a vector is returned.
- If `d = 'row'`, the scanning left iteration runs on the rows of the matrix `mat` and a matrix is returned.
- If `d = 'col'`, the scanning left iteration runs on the columns of the matrix `mat` and a matrix is returned.
- If `d = 'diag'`, the scanning left iteration runs on the diagonal of the matrix `mat` and a vector is returned.

Note that ommitting both `x0` and `d` implies to not specify `r` as well, otherwise the latter will be interpreted as `x0`. If `r = nil`, then the type of the returned matrix is determined by the type of the value returned by `f()` called once before scanning. Default: `x0 = mat[1]` (or first row or column element), `d = 'vec'`.

`mat:`**`scanr`**(*f*, *[x0_,] [d_,] r_*)

Same as *mat:scanl()* but the *callable* (or the operator string) `f` is applied iteratively using the scanning right (backward with decreasing indexes) expression `v = f(mat[n], v)`. Default: `x0 = mat[#mat]` (or last row or column element), `d = 'vec'`.

5.8 Mapping Real-like Methods

The following table lists the methods built from the application of *mat:map()* and variants to the real-like functions from the module `MAD.gmath` for *matrix* and *cmatrix*. The methods `mat:sign()`, `mat:sign1()` and `mat:atan2()` are not available for *cmatrix*, and only the methods `mat:abs()`, `mat:sqr()` and `mat:sign()` are available for *imatrix*.

Functions	Equivalent Mapping
mat:abs(r_)	mat:map(abs,r_)
mat:acos(r_)	mat:map(acos,r_)
mat:acosh(r_)	mat:map(acosh,r_)
mat:acot(r_)	mat:map(acot,r_)
mat:acoth(r_)	mat:map(acoth,r_)
mat:asin(r_)	mat:map(asin,r_)
mat:asinh(r_)	mat:map(asinh,r_)
mat:asinc(r_)	mat:map(asinc,r_)
mat:asinhc(r_)	mat:map(asinhc,r_)
mat:atan(r_)	mat:map(atan,r_)
mat:atan2(y,r_)	mat:map2(y,atan2,r_)
mat:atanh(r_)	mat:map(atanh,r_)
mat:ceil(r_)	mat:map(ceil,r_)
mat:cos(r_)	mat:map(cos,r_)
mat:cosh(r_)	mat:map(cosh,r_)
mat:cot(r_)	mat:map(cot,r_)
mat:coth(r_)	mat:map(coth,r_)
mat:exp(r_)	mat:map(exp,r_)
mat:floor(r_)	mat:map(floor,r_)
mat:frac(r_)	mat:map(frac,r_)
mat:hypot(y,r_)	mat:map2(y,hypot,r_)
mat:hypot3(y,z,r_)	mat:map3(y,z,hypot3,r_)
mat:invsqrt([v_,]r_)	mat:map2(v_ or 1,invsqrt,r_)
mat:log(r_)	mat:map(log,r_)
mat:log10(r_)	mat:map(log10,r_)
mat:round(r_)	mat:map(round,r_)
mat:sign(r_)	mat:map(sign,r_)
mat:sign1(r_)	mat:map(sign1,r_)
mat:sin(r_)	mat:map(sin,r_)
mat:sinc(r_)	mat:map(sinc,r_)
mat:sinh(r_)	mat:map(sinh,r_)
mat:sinhc(r_)	mat:map(sinhc,r_)
mat:sqr(r_)	mat:map(sqr,r_)
mat:sqrt(r_)	mat:map(sqrt,r_)
mat:tan(r_)	mat:map(tan,r_)
mat:tanh(r_)	mat:map(tanh,r_)
mat:trunc(r_)	mat:map(trunc,r_)

5.9 Mapping Complex-like Methods

The following table lists the methods built from the application of *mat:map()* to the the complex-like functions from the module MAD.gmath for *matrix* and *cmatrix*.

Functions	Equivalent Mapping
mat:cabs(r_)	mat:map(cabs,r_)
mat:carg(r_)	mat:map(carg,r_)
mat:conj(r_)	mat:map(conj,r_)
mat:cplx(im_,r_)	mat:map2(im_, cplx, r_)
mat:fabs(r_)	mat:map(fabs,r_)
mat:imag(r_)	mat:map(imag,r_)
mat:polar(r_)	mat:map(polar,r_)
mat:proj(r_)	mat:map(proj,r_)
mat:real(r_)	mat:map(real,r_)
mat:rect(r_)	mat:map(rect,r_)
mat:reim(re_, im_)	mat:real(re_), mat:imag(im_)

The method mat:cplx() has a special implementation that allows to used it without a real part, e.g. im.cplx(nil, im, r_).

The method mat:conjugate() is also available as an alias for mat:conj().

5.10 Mapping Error-like Methods

The following table lists the methods built from the application of *mat:map()* to the error-like functions from the module MAD.gmath for *matrix* and *cmatrix*.

Functions	Equivalent Mapping
mat:erf([rtol_,]r_)	mat:map2(rtol_,erf,r_)
mat:erfc([rtol_,]r_)	mat:map2(rtol_,erfc,r_)
mat:erfcx([rtol_,]r_)	mat:map2(rtol_,erfcx,r_)
mat:erfi([rtol_,]r_)	mat:map2(rtol_,erfi,r_)
mat:wf([rtol_,]r_)	mat:map2(rtol_,wf,r_)

5.11 Mapping Vector-like Methods

The following table lists the methods built from the application of *mat:map2()* to the vector-like functions from the module MAD.gfunc for *matrix*, *cmatrix*, and *imatrix*.

Functions	Equivalent Mapping
mat:emul(mat2,r_)	mat:map2(mat2,mul,r_)
mat:ediv(mat2,r_)	mat:map2(mat2,div,r_)
mat:emod(mat2,r_)	mat:map2(mat2,mod,r_)
mat:epow(mat2,r_)	mat:map2(mat2,pow,r_)

5.12 Folding Methods

The following table lists the methods built from the application of *mat:foldl()* to the functions from the module MAD.gmath for *matrix*, *cmatrix*, and *imatrix*. The methods mat:min() and mat:max() are not available for *cmatrix*.

Functions	Equivalent Folding
mat:all(p,d_,r_)	mat:foldl(all(p),false,d_,r_)
mat:any(p,d_,r_)	mat:foldl(any(p),true,d_,r_)
mat:min(d_,r_)	mat:foldl(min,nil,d_,r_)
mat:max(d_,r_)	mat:foldl(max,nil,d_,r_)
mat:sum(d_,r_)	mat:foldl(add,nil,d_,r_)
mat:prod(d_,r_)	mat:foldl(mul,nil,d_,r_)
mat:sumsqr(d_,r_)	mat:foldl(sumsqrl,0,d_,r_)
mat:sumabs(d_,r_)	mat:foldl(sumabsl,0,d_,r_)
mat:minabs(d_,r_)	mat:foldl(minabsl,inf,d_,r_)
mat:maxabs(d_,r_)	mat:foldl(maxabsl,0,d_,r_)

Where any() and all() are functions that bind the predicate p to the propagation of the logical AND and the logical OR respectively, that can be implemented like:

```
- all = \p -> \r,x -> lbool(land(r, p(x)))
- any = \p -> \r,x -> lbool(lor (r, p(x)))
```

5.13 Scanning Methods

The following table lists the methods built from the application of *mat:scanl()* and *mat:scanr()* to the functions from the module MAD.gmath for *matrix* and *cmatrix*. The methods mat:accmin(), mat:raccmin(), mat:accmax() and mat:raccmax() are not available for *cmatrix*.

Functions	Equivalent Scanning
mat:accmin(d_,r_)	mat:scanl(min,nil,d_,r_)
mat:accmax(d_,r_)	mat:scanl(max,nil,d_,r_)
mat:accsum(d_,r_)	mat:scanl(add,nil,d_,r_)
mat:accprod(d_,r_)	mat:scanl(mul,nil,d_,r_)
mat:accsumsqr(d_,r_)	mat:scanl(sumsqrl,0,d_,r_)
mat:accsumabs(d_,r_)	mat:scanl(sumabsl,0,d_,r_)
mat:accminabs(d_,r_)	mat:scanl(minabsl,inf,d_,r_)
mat:accmaxabs(d_,r_)	mat:scanl(maxabsl,0,d_,r_)
mat:raccmin(d_,r_)	mat:scanr(min,nil,d_,r_)
mat:raccmax(d_,r_)	mat:scanr(max,nil,d_,r_)
mat:raccsum(d_,r_)	mat:scanr(add,nil,d_,r_)
mat:raccprod(d_,r_)	mat:scanr(mul,nil,d_,r_)
mat:raccsumsqr(d_,r_)	mat:scanr(sumsqrr,0,d_,r_)
mat:raccsumabs(d_,r_)	mat:scanr(sumabsr,0,d_,r_)
mat:raccminabs(d_,r_)	mat:scanr(minabsr,inf,d_,r_)
mat:raccmaxabs(d_,r_)	mat:scanr(maxabsr,0,d_,r_)

The method `mat:accumulate()` is also available as an alias for `mat:accsum()`.

5.14 Matrix Functions

The following table lists the methods built from the application of *mat:mfun()* to the real-like functions from the module `MAD.gmath` for *matrix* and *cmatrix*.

Functions	Equivalent Matrix Function
mat:macos()	mat:mfun(acos)
mat:macosh()	mat:mfun(acosh)
mat:macot()	mat:mfun(acot)
mat:macoth()	mat:mfun(acoth)
mat:masin()	mat:mfun(asin)
mat:masinh()	mat:mfun(asinh)
mat:masinc()	mat:mfun(asinc)
mat:masinhc()	mat:mfun(asinhc)
mat:matan()	mat:mfun(atan)
mat:matanh()	mat:mfun(atanh)
mat:mcos()	mat:mfun(cos)
mat:mcosh()	mat:mfun(cosh)
mat:mcot()	mat:mfun(cot)
mat:mcoth()	mat:mfun(coth)
mat:mexp()	mat:mfun(exp)
mat:mlog()	mat:mfun(log)
mat:mlog10()	mat:mfun(log10)
mat:msin()	mat:mfun(sin)
mat:msinc()	mat:mfun(sinc)
mat:msinh()	mat:mfun(sinh)
mat:msinhc()	mat:mfun(sinhc)
mat:msqrt()	mat:mfun(sqrt)
mat:mtan()	mat:mfun(tan)
mat:mtanh()	mat:mfun(tanh)

5.15 Operator-like Methods

mat:**unm**(*r_*)

Equivalent to -mat with the possibility to place the result in r.

mat:**add**(*a*, *r_*)

Equivalent to mat + a with the possibility to place the result in r.

mat:**sub**(*a*, *r_*)

Equivalent to mat - a with the possibility to place the result in r.

mat:**mul**(*a*, *r_*)

Equivalent to mat * a with the possibility to place the result in r.

mat:**tmul**(*mat2*, *r_*)

Equivalent to mat:t() * mat2 with the possibility to place the result in r.

mat:**mult**(*mat2*, *r_*)

Equivalent to mat * mat2:t() with the possibility to place the result in r.

`mat:`**`dmul`**`(`*mat2*, *r_*`)`

Equivalent to `mat:getdiag():diag() * mat2` with the possibility to place the result in `r`. If `mat` is a vector, it will be interpreted as the diagonal of a square matrix like in *mat:diag()*, i.e. omitting *mat:getdiag()* is the previous expression.

`mat:`**`muld`**`(`*mat2*, *r_*`)`

Equivalent to `mat * mat2:getdiag():diag()` with the possibility to place the result in `r`. If `mat2` is a vector, it will be interpreted as the diagonal of a square matrix like in `mat2:diag()`, i.e. omitting `mat2:getdiag()` is the previous expression.

`mat:`**`div`**`(`*a*, *[r_,] rcond_*`)`

Equivalent to `mat / a` with the possibility to place the result in `r`, and to specify the conditional number `rcond` used by the solver to determine the effective rank of non-square systems. Default: `rcond = eps`.

`mat:`**`inv`**`(`*[r_,] rcond_*`)`

Equivalent to `mat.div(1, mat, r_, rcond_)`.

`mat:`**`pow`**`(`*n*, *r_*`)`

Equivalent to `mat ^ n` with the possibility to place the result in `r`.

`mat:`**`eq`**`(`*a*, *tol_*`)`

Equivalent to `mat == a` with the possibility to specify the tolerance `tol` of the comparison. Default: `tol_ = 0`.

`mat:`**`concat`**`(`*mat2*, *[d_,] r_*`)`

Equivalent to `mat .. mat2` with the possibility to place the result in `r` and to specify the direction of the concatenation:

- If `d = 'vec'`, it concatenates the matrices (appended as vectors)
- If `d = 'row'`, it concatenates the rows (horizontal)
- If `d = 'col'`, it concatenates the columns (vectical)

Default: `d_ = 'row'`.

5.16 Special Methods

`mat:`**`transpose`**`(`*[c_,] r_*`)`

`mat:`**`t`**`(`*[c_,] r_*`)`

Return a real, complex or integer matrix or `r` resulting from the conjugate transpose M^* of the matrix `mat` unless `c = false` which disables the conjugate to get M^τ. If `r = 'in'` then it is assigned `mat`.

`mat:`**`trace`**`()`

`mat:`**`tr`**`()`

Return the Trace of the real or complex `mat` equivalent to `mat:sum('diag')`.

`mat:`**`inner`**`(`*mat2*`)`

`mat:`**`dot`**`(`*mat2*`)`

Return the Inner Product of the two real or complex matrices `mat` and `mat2` with compatible sizes, i.e. return $M^*.M_2$ interpreting matrices as vectors. Note that multiple dot products, i.e. not interpreting matrices as vectors, can be achieved with *`mat:tmul()`*.

`mat:`**`outer`**`(`*mat2*`, `*r_*`)`

Return the real or complex matrix resulting from the Outer Product of the two real or complex matrices `mat` and `mat2`, i.e. return $M.M_2^*$ interpreting matrices as vectors.

`mat:`**`cross`**`(`*mat2*`, `*r_*`)`

Return the real or complex matrix resulting from the Cross Product of the two real or complex matrices `mat` and `mat2` with compatible sizes, i.e. return $M \times M_2$ interpreting matrices as a list of $[3\times 1]$ column vectors.

`mat:`**`mixed`**`(`*mat2*`, `*mat3*`, `*r_*`)`

Return the real or complex matrix resulting from the Mixed Product of the three real or complex matrices `mat`, `mat2` and `mat3` with compatible sizes, i.e. return $M^*.(M_2 \times M_3)$ interpreting matrices as a list of $[3 \times 1]$ column vectors.

`mat:`**`norm`**`()`

Return the Frobenius norm of the matrix $\|M\|_2$. Other L_p matrix norms and variants can be easily calculated using already provided methods, e.g. L_1 = `mat:sumabs('col'):max()`, L_∞ = `mat:sumabs('row'):max()`, and L_2 = `mat:svd():max()`.

`mat:`**`dist`**`(`*mat2*`)`

Equivalent to `(mat - mat2):norm()`.

`mat:`**`unit`**`()`

Return the scaled matrix `mat` to the unit norm equivalent to `mat:div(mat:norm(), mat)`.

`mat:`**`center`**`(`*d_*`)`

Return the centered matrix `mat` to have zero mean equivalent to `mat:sub(mat:mean(),mat)`. The direction `d` indicates how the centering must be performed:

- If `d = 'vec'`, it centers the entire matrix by substracting its mean.
- If `d = 'row'`, it centers each row by substracting their mean.
- If `d = 'col'` , it centers each column by substracting their mean.
- If `d = 'diag'`, it centers the diagonal by substracting its mean.

Default: `d_ = 'vec'`.

`mat:`**`angle`**`(`*mat2*`, `*n_*`)`

Return the angle between the two real or complex vectors `mat` and `mat2` using the method *`mat:inner()`*. If `n` is provided, the sign of `mat:mixed(mat2, n)` is used to define the angle in $[-\pi, \pi]$, otherwise it is defined in $[0, \pi]$.

`mat:`**`minmax`**`(`*abs_*`)`

Return the minimum and maximum values of the elements of the real, complex or integer matrix `mat`. If `abs = true`, it returns the minimum and maximum absolute values of the elements. Default: `abs_ = false`.

`mat:`**`iminmax`**`(`*abs_*`)`

Return the two vector-like indexes of the minimum and maximum values of the elements of the real, complex or integer matrix `mat`. If `abs = true`, it returns the indexes of the minimum and maximum absolute values of the elements. Default: `abs_ = false`.

`mat:`**`mean`**`()`

Equivalent to `mat:sum()/#mat`, i.e. interpreting the matrix as a vector.

`mat:`**`variance`**`()`

Equivalent to `(mat - mat:mean()):sumsqr()/(#mat-1)`, i.e. return the unbiased estimator of the variance with second order Bessel's correction, interpreting the matrix as a vector.

`mat:`**`ksum`**`()`

`mat:`**`kdot`**`(`*mat2*`)`

Same as `mat:sum()` and *`mat:dot()`* respectively, except that they use the more accurate Kahan Babushka Neumaier algorithm for the summation, e.g. the sum of the elements of the vector $[1, 10^{100}, 1, -10^{100}]$ should return 0 with `sum()` and the correct answer 2 with `ksum()`.

`mat:`**`kadd`**`(`*a*, *x*`)`

Return the real or complex matrix `mat` filled with the linear combination of the compatible matrices stored in `x` times the scalars stored in `a`, i.e. `mat = a[1]*x[1] + a[2]*x[2] ...`

`mat:`**`eval`**`(`*x0*`)`

Return the evaluation of the real or complex matrix `mat` at the value `x0`, i.e. interpreting the matrix as a vector of polynomial coefficients of increasing orders in `x` evaluated at `x = x0` using Horner's method.

`mat:`**`sympconj`**`(`*r_*`)`

`mat:`**`bar`**`(`*r_*`)`

Return a real or complex matrix or `r` resulting from the symplectic conjugate of the matrix `mat`, with $\bar{M} = -S_{2n} M^* S_{2n}$, and $M^{-1} = \bar{M}$ if M is symplectic. If `r = 'in'` then it is assigned `mat`.

`mat:`**`symperr`**`(`*r_*`)`

Return the norm of the symplectic deviation matrix given by $M^* S_{2n} M - S_{2n}$ of the real or complex matrix `mat`. If `r` is provided, it is filled with the symplectic deviation matrix.

`mat:`**`dif`**`(`*mat2*, *r_*`)`

Return a real or complex matrix or `r` resulting from the term-by-term difference between the matrices `mat` and `mat2` using the absolute difference for values with magnitude below 1 and the relative difference otherwise, i.e. $r_i = (x_i - y_i)/\max(|x_i|, 1)$.

5.17 Solvers and Decompositions

Except for `nsolve()`, the solvers hereafter are wrappers around the library Lapack[2].

`mat:`**`solve`**`(`*b*, *rcond_*`)`

Return the real or complex $[n \times p]$ matrix x as the minimum-norm solution of the linear least square problem $\min \|Ax - B\|$ where A is the real or complex $[m \times n]$ matrix `mat` and B is a $[m \times p]$ matrix `b` of the same type as `mat`, using LU, QR or LQ factorisation depending on the shape of the system. The conditional number `rcond` is used by the solver to determine the effective rank of non-square system. This method also returns the rank of the system. Default: `rcond_ = eps`.

`mat:`**`ssolve`**`(`*b*, *rcond_*`)`

Return the real or complex $[n \times p]$ matrix x as the minimum-norm solution of the linear least square problem $\min \|Ax - B\|$ where A is the real or complex $[m \times n]$ matrix `mat` and B is a $[m \times p]$ matrix `b` of the same type as `mat`, using SVD factorisation. The conditional number `rcond` is used by the solver to determine the effective rank of the system. This method also returns the rank of the system followed by the real $[\min(m, n) \times 1]$ vector of singluar values. Default: `rcond_ = eps`.

`mat:`**`gsolve`**`(`*b*, *c*, *d*`)`

Return the real or complex $[n \times 1]$ vector `x` as the minimum-norm solution of the linear least square problem $\min \|Ax - C\|$ under the constraint $Bx = D$ where A is the real or complex $[m \times n]$ matrix `mat`, B is a $[p \times n]$ matrix `b`, C is a $[m \times 1]$ vector `c` and D is a $[p \times 1]$ vector `d`, all of the same type as `mat`, using QR or LQ factorisation depending on the shape of the system. This method also returns the norm of the residues and the status `info`.

`mat:`**`gmsolve`**`(`*b*, *d*`)`

Return the real or complex $[n \times 1]$ vector `x` and $[p \times 1]$ matrix `y` as the minimum-norm solution of the linear Gauss-Markov problem $\min_x \|y\|$ under the constraint $Ax + By = D$ where A is the $[m \times n]$ real or complex matrix `mat`, B is a $[m \times p]$ matrix `b`, and D is a $[m \times 1]$ vector `d`, both of the same type as `mat`, using QR or LQ factorisation depending on the shape of the system. This method also returns the status `info`.

`mat:`**`nsolve`**`(`*b*, *nc_*, *tol_*`)`

Return the real $[n \times 1]$ vector `x` (of correctors kicks) as the minimum-norm solution of the linear (best-kick) least square problem $\min \|Ax - B\|$ where A is the real $[m \times n]$ (response) matrix `mat` and B is a real $[m \times 1]$ vector `b` (of monitors readings), using the MICADO[3] algorithm based on the Householder-Golub method [MICADO]. The argument `nc` is the maximum number of correctors to use with $0 < n_c \leq n$ and the argument `tol` is a convergence threshold (on the residues) to stop the (orbit) correction if $\|Ax - B\| \leq m \times$ `tol`. This method also returns the updated number of correctors n_c effectively used during the correction followed by the real $[m \times 1]$ vector of residues. Default: `nc_ = ncol`, `tol_ = eps`.

[2] The solvers are based, among others, on the following Lapack drivers:
- `dgesv()` and `zgesv()` for LU factorization.
- `dgelsy()` and `zgelsy()` for QR or LQ factorization.
- `dgelsd()` and `zgelsd()` for SVD factorisation.
- `dgees()` and `zgees()` for Schur factorisation.
- `dgglse()` and `zgglse()` for equality-constrained linear Least Squares problems.
- `dggglm()` and `zggglm()` for general Gauss-Markov linear model problems.

[3] MICADO stands for "Minimisation des CArrés des Distortions d'Orbite" in french.

`mat:`**`pcacnd`**(*ns_*, *rcond_*)

Return the integer column vector `ic` containing the indexes of the columns to remove from the real or complex $[m \times n]$ matrix `mat` using the Principal Component Analysis. The argument `ns` is the maximum number of singular values to consider and `rcond` is the conditioning number used to select the singular values versus the largest one, i.e. consider the `ns` larger singular values σ_i such that $\sigma_i > \sigma_{\max} \times$`rcond`. This method also returns the real $[\min(m,n) \times 1]$ vector of singluar values. Default: `ns_ = ncol`, `rcond_ = eps`.

`mat:`**`svdcnd`**(*ns_*, *rcond_*, *tol_*)

Return the integer column vector `ic` containing the indexes of the columns to remove from the real or complex $[m \times n]$ matrix `mat` based on the analysis of the right matrix V from the SVD decomposition USV. The argument `ns` is the maximum number of singular values to consider and `rcond` is the conditioning number used to select the singular values versus the largest one, i.e. consider the `ns` larger singular values σ_i such that $\sigma_i > \sigma_{\max} \times$`rcond`. The argument `tol` is a threshold similar to `rcond` used to reject components in V that have similar or opposite effect than components already encountered. This method also returns the real $[\min(m,n) \times 1]$ vector of singluar values. Default: `ns_ = min(nrow,ncol)`, `rcond_ = eps`.

`mat:`**`svd`**`()`

Return the real vector of the singular values and the two real or complex matrices resulting from the SVD factorisation of the real or complex matrix `mat`, followed the status `info`. The singular values are positive and sorted in decreasing order of values, i.e. largest first.

`mat:`**`eigen`**(*vr_*, *vl_*)

Return the complex vector filled with the eigenvalues followed by the by the status `info` and the two optional real or complex matrices `vr` and `vl` containing the right and the *transposed* left eigenvectors resulting from the Eigen Decomposition of the real or complex square matrix `mat`. The eigenvectors are normalized to have unit Euclidean norm and their largest component real, and satisfy $Av_r = \lambda v_r$ and $v_l^\tau A = \lambda v_l^\tau$.

`mat:`**`det`**`()`

Return the Determinant of the real or complex square matrix `mat` using LU factorisation for better numerical stability, followed by the status `info`.

`mat:`**`mfun`**(*fun*)

Return the real or complex matrix resulting from the matrix function `fun` applied to the real or complex matrix `mat`. So far, *`mat:mfun()`* uses the eigen decomposition of the matrix `mat`, which must be Diagonalizable. See the section *Matrix Functions* for the list of matrix functions already provided. Future versions of this method may be extended to use the more general Schur-Parlett algorithm [MATFUN], and other specialized versions for `msqrt()`, `mpow`, `mexp`, and `mlog` may be implemented too.

5.18 Fourier Transforms and Convolutions

The methods described is this section are based on the FFTW and NFFT libraries.

`mat:`**`fft`**`(`*[d_,] r_*`)`

Return the complex $[n_r \times n_c]$ vector, matrix or `r` resulting from the 1D or 2D Fourier Transform of the real or complex $[n_r \times n_c]$ vector or matrix `mat` in the direction given by `d`:

- If `d = 'vec'`, it returns a 1D vector FFT of length $n_r n_c$.
- If `d = 'row'`, it returns n_r 1D row FFTs of length n_c.
- If `d = 'col'`, it returns n_c 1D column FFTs of length n_r.
- otherwise, it returns a 2D FFT of sizes $[n_r \times n_c]$.

`mat:`**`ifft`**`(`*[d_,] r_*`)`

Return the complex $[n_r \times n_c]$ vector, matrix or `r` resulting from the 1D or 2D inverse Fourier Transform of the complex $[n_r \times n_c]$ vector or matrix `mat`. See *`mat:fft()`* for the direction `d`.

`mat:`**`rfft`**`(`*[d_,] r_*`)`

Return the complex $[n_r \times \lfloor n_c/2+1 \rfloor]$ vector, matrix or `r` resulting from the 1D or 2D Fourier Transform of the *real* $[n_r \times n_c]$ vector or matrix `mat`. This method used an optimized version of the FFT for real data, which is about twice as fast and compact as the method *`mat:fft()`*. See *`mat:fft()`* for the direction `d`.

`mat:`**`irfft`**`(`*[d_,] r*`)`

Return the *real* $[n_r \times n_c]$ vector, matrix or `r` resulting from the 1D or 2D inverse Fourier Transform of the complex $[n_r \times \lfloor n_c/2+1 \rfloor]$ vector or matrix `mat` as computed by the method *`mat:rfft()`*. See *`mat:fft()`* for the direction `d`. Note that `r` must be provided to specify the correct n_c of the result.

`mat:`**`nfft`**`(`*p_, r_*`)`

Return the complex vector, matrix or `r` resulting from the 1D or 2D *Nonequispaced* Fourier Transform of the real or complex vector or matrix `mat` respectively at `p` time nodes.

`mat:`**`infft`**`(`*p_, r_*`)`

Return the complex vector, matrix or `r` resulting from the 1D or 2D *Nonequispaced* inverse Fourier Transform of the real or complex vector or matrix `mat` respectively at `p` frequency nodes.

`mat:`**`conv`**`(`*[y_,] [d_], r_*`)`

Return the real or complex vector, matrix or `r` resulting from the 1D or 2D Convolution between the compatible real or complex vectors or matrices `mat` and `y` respectively. See *`mat:fft()`* for the direction `d`. Default: `y = mat`.

`mat:`**`corr`**`(`*[y_,] [d_], r_*`)`

Return the real or complex vector, matrix or `r` resulting from the 1D or 2D Correlation between the compatible real or complex vectors or matrices `mat` and `y` respectively. See *`mat:fft()`* for the direction `d`. Default: `y = mat`.

`mat:`**`covar`**`(`*[y_,] [d_,] r_*`)`

Return the real or complex vector, matrix or `r` resulting from the 1D or 2D Covariance between the compatible real or complex vectors or matrices `mat` and `y` respectively. See *`mat:fft()`* for the direction `d`. Default: `y = mat`.

`mat:`**`zpad`**(*nr*, *nc*, *d_*)

Return the real or complex vector or matrix resulting from the zero padding of the matrix `mat` extended to the sizes `nr` and `nc`, following the direction `d`:

- If `d = 'vec'`, it pads the zeros at the end of the matrix equivalent `x:same(nr,nc):setvec(1..#x,x)`, i.e. interpreting the matrix as a vector.
- If `d = 'row'`, it pads the zeros at the end of the rows equivalent `x:same(x.nrow,nc):setsub(1..x.nrow,1..x.ncol,x)`, i.e. ignoring `nr`.
- If `d = 'col'`, it pads the zeros at the end of the columns equivalent `x:same(nr,x.ncol):setsub(1..x.nrow,1..x.ncol,x)`, i.e. ignoring `nc`.
- otherwise, it pads the zeros at the end of the rows and the columns equivalent to `x:same(nr,nc):setsub(1..x.nrow,1..x.ncol,x)`.

If the zero padding does not change the size of `mat`, the orignal `mat` is returned unchanged.

5.19 Rotations

This section describe methods dealing with 2D and 3D rotations (see Rotation Matrix) with angles in radians and trigonometric (counter-clockwise) direction for a right-handed frame, and where the following convention hold: `ax = -phi` is the *elevation* angle, `ay = theta` is the *azimuthal* angle and `az = psi` is the *roll/tilt* angle.

`mat:`**`rot`**(*a*)

Return the $[2 \times 2]$ real *matrix* `mat` filled with a 2D rotation of angle `a`.

`mat:`**`rotx`**(*a*)

`mat:`**`roty`**(*a*)

`mat:`**`rotz`**(*a*)

Return the $[3 \times 3]$ real *matrix* `mat` filled with a 3D rotation of angle `a` around the x-axis, y-axis and z-axis respectively.

`mat:`**`rotxy`**(*ax*, *ay*, *inv_*)

`mat:`**`rotxz`**(*ax*, *az*, *inv_*)

`mat:`**`rotyx`**(*ay*, *ax*, *inv_*)

`mat:`**`rotyz`**(*ay*, *az*, *inv_*)

`mat:`**`rotzx`**(*az*, *ax*, *inv_*)

`mat:`**`rotzy`**(*az*, *ay*, *inv_*)

Return the $[3 \times 3]$ real *matrix* `mat` filled with a 3D rotation of the first angle argument `ax`, `ay` or `az` around the x-axis, y-axis or z-axis respectively *followed* by another 3D rotation of the second angle argument `ax`, `ay` or `az` around the x-axis, y-axis or z-axis respectively of the frame rotated by the first rotation. If `inv` is true, the returned matrix is the inverse rotation, i.e. the transposed matrix.

`mat:`**`rotxyz`**(*ax*, *ay*, *az*, *inv_*)

`mat:`**`rotxzy`**(*ax*, *az*, *ay*, *inv_*)

`mat:`**`rotyxz`**(*ay*, *ax*, *az*, *inv_*)

`mat:`**`rotyzx`**(*ay*, *az*, *ax*, *inv_*)

`mat:`**`rotzxy`**(*az*, *ax*, *ay*, *inv_*)

`mat:`**`rotzyx`**(*az*, *ay*, *ax*, *inv_*)

Return the $[3 \times 3]$ real *matrix* `mat` filled with a 3D rotation of the first angle argument `ax`, `ay` or `az` around the x-axis, y-axis or z-axis respectively *followed* by another 3D rotation of the second angle argument `ax`, `ay` or `az` around the x-axis, y-axis or z-axis respectively of the frame rotated by the first rotation, and *followed* by a last 3D rotation of the third angle argument `ax`, `ay` or `az` around the x-axis, y-axis or z-axis respectively of the frame already rotated by the two first rotations. If `inv` is true, the returned matrix is the inverse rotations, i.e. the transposed matrix.

`mat:`**`torotxyz`**(*inv_*)

`mat:`**`torotxzy`**(*inv_*)

`mat:`**`torotyxz`**(*inv_*)

`mat:`**`torotyzx`**(*inv_*)

`mat:`**`torotzxy`**(*inv_*)

`mat:`**`torotzyx`**(*inv_*)

Return three real *number* representing the three angles `ax`, `ay` and `az` (always in this order) of the 3D rotations stored in the $[3 \times 3]$ real *matrix* `mat` by the methods with corresponding names. If `inv` is true, the inverse rotations are returned, i.e. extracted from the transposed matrix.

`mat:`**`rotv`**(*v*, *av*, *inv_*)

Return the $[3 \times 3]$ real *matrix* `mat` filled with a 3D rotation of angle `av` around the axis defined by the 3D vector-like `v` (see Axis-Angle representation). If `inv` is true, the returned matrix is the inverse rotation, i.e. the transposed matrix.

`mat:`**`torotv`**(*v_*, *inv_*)

Return a real *number* representing the angle of the 3D rotation around the axis defined by a 3D vector as stored in the $[3 \times 3]$ real *matrix* `mat` by the method *`mat:rotv()`*. If the *iterable* `v` is provided, it is filled with the components of the unit vector that defines the axis of the rotation. If `inv` is true, the inverse rotation is returned, i.e. extracted from the transposed matrix.

`mat:`**`rotq`**(*q*, *inv_*)

Return the $[3 \times 3]$ real *matrix* `mat` filled with a 3D rotation defined by the quaternion `q` (see Axis-Angle representation). If `inv` is true, the returned matrix is the inverse rotation, i.e. the transposed matrix.

`mat:`**`torotq`**(*q_*, *inv_*)

Return a quaternion representing the 3D rotation as stored in the $[3 \times 3]$ real *matrix* `mat` by the method *`mat:rotq()`*. If the *iterable* `q` is provided, it is filled with the components of the quaternion otherwise the quaternion is returned in a *list* of length 4. If `inv` is true, the inverse rotation is returned, i.e. extracted from the transposed matrix.

5.20 Conversions

`mat:`**`tostring`**(*sep_*, *lsep_*)

Return the string containing the real, complex or integer matrix converted to string. The argument `sep` and `lsep` are used as separator for columns and rows respectively. The elements values are formated using *`tostring()`* that follows the `option.numfmt` string format for real numbers. Default: `sep = " "`, `lsep = "\n"`.

mat:**totable**(*[d_,] r_*)

Return the table or r containing the real, complex or integer matrix converted to tables, i.e. one per row unless mat is a vector or the direction d = 'vec'.

5.21 Input and Output

mat:**write**(*filename_, name_, eps_, line_, nl_*)

Return the real, complex or integer matrix after writing it to the file filename opened with MAD.utility.openfile(). The content of the matrix mat is preceded by a header containing enough information to read it back. If name is provided, it is part of the header. If line = 'line', the matrix is displayed on a single line with rows separated by a semicolumn, otherwise it is displayed on multiple lines separated by nl. Elements with absolute value below eps are displayed as zeros. The formats defined by MAD.option.numfmt and MAD.option.intfmt are used to format numbers of *matrix*, *cmatrix* and *imatrix* respectively. Default: filename_ = io.stdout, name_ = '', eps_ = 0, line_ = nil, nl_ = '\n'.

mat:**print**(*name_, eps_, line_, nl_*)

Equivalent to mat:write(nil, name_, eps_, line_, nl_).

mat:**read**(*filename_*)

Return the real, complex or integer matrix read from the file filename opened with MAD.utility.openfile(). Note that the matrix mat is only used to call the method :read() and has no impact on the type and sizes of the returned matrix fully characterized by the content of the file. Default: filename_ = io.stdin.

6 Operators

#mat

Return the size of the real, complex or integer matrix mat, i.e. the number of elements interpreting the matrix as a vector.

mat[n]

Return the value of the element at index n of the real, complex or integer matrix mat for 1 <= n <= #mat, i.e. interpreting the matrix as a vector, nil otherwise.

mat[n] = v

Assign the value v to the element at index n of the real, complex or integer matrix mat for 1 <= n <= #mat, i.e. interpreting the matrix as a vector, otherwise raise an *"out of bounds"* error.

-mat

Return a real, complex or integer matrix resulting from the unary minus applied individually to all elements of the matrix mat.

num + mat

mat + num

mat + mat2

Return a *matrix* resulting from the sum of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

num + cmat

cpx + mat

cpx + cmat

mat + cpx

mat + cmat

cmat + num

cmat + cpx

cmat + mat

cmat + cmat2

Return a *cmatrix* resulting from the sum of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

idx + imat

imat + idx

imat + imat

Return a *imatrix* resulting from the sum of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

num - mat

mat - num

mat - mat2

Return a *matrix* resulting from the difference of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

num - cmat

cpx - mat

cpx - cmat

mat - cpx

mat - cmat

cmat - num

cmat - cpx

cmat - mat

cmat - cmat2

Return a *cmatrix* resulting from the difference of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

idx - imat

imat - idx

imat - imat

Return a *imatrix* resulting from the difference of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

num * mat

mat * num

mat * mat2

Return a *matrix* resulting from the product of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix. If the two operands are matrices, the mathematical matrix multiplication is performed.

num * cmat

cpx * mat

cpx * cmat

mat * cpx

mat * cmat

cmat * num

cmat * cpx

cmat * mat

cmat * cmat2

Return a *cmatrix* resulting from the product of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix. If the two operands are matrices, the mathematical matrix multiplication is performed.

idx * imat

imat * idx

Return a *imatrix* resulting from the product of the left and right operands that must have compatible sizes. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

num / mat

mat / num

mat / mat2

Return a *matrix* resulting from the division of the left and right operands that must have compatible sizes. If the right operand is a scalar, the operator will be applied individually to all elements of the matrix. If the left operand is a scalar the operation `x/Y` is converted to `x (I/Y)` where `I` is the identity matrix with compatible sizes. If the right operand is a matrix, the operation `X/Y` is performed using a system solver based on LU, QR or LQ factorisation depending on the shape of the system.

num / cmat

cpx / mat

cpx / cmat

mat / cpx

mat / cmat

cmat / num

cmat / cpx

cmat / mat

cmat / cmat2

Return a *cmatrix* resulting from the division of the left and right operands that must have compatible sizes. If the right operand is a scalar, the operator will be applied individually to all elements of the matrix. If the left operand is a scalar the operation `x/Y` is converted to `x (I/Y)` where `I` is the identity matrix with compatible sizes. If the right operand is a matrix, the operation `X/Y` is performed using a system solver based on LU, QR or LQ factorisation depending on the shape of the system.

imat / idx

Return a *imatrix* resulting from the division of the left and right operands, where the operator will be applied individually to all elements of the matrix.

mat % num

mat % mat

Return a *matrix* resulting from the modulo between the elements of the left and right operands that must have compatible sizes. If the right operand is a scalar, the operator will be applied individually to all elements of the matrix.

cmat % num

cmat % cpx

cmat % mat

cmat % cmat

Return a *cmatrix* resulting from the modulo between the elements of the left and right operands that must have compatible sizes. If the right operand is a scalar, the operator will be applied individually to all elements of the matrix.

imat % idx

imat % imat

Return a *imatrix* resulting from the modulo between the elements of the left and right operands that must have compatible sizes. If the right operand is a scalar, the operator will be applied individually to all elements of the matrix.

mat ^ n

cmat ^ n

Return a *matrix* or *cmatrix* resulting from `n` products of the square input matrix by itself. If `n` is negative, the inverse of the matrix is used for the product.

num == mat

num == cmat

num == imat

cpx == mat

cpx == cmat

mat == num

mat == cpx

mat == mat2

mat == cmat

cmat == num

cmat == cpx

cmat == mat

cmat == cmat2

imat == num

imat == imat2

Return `false` if the left and right operands have incompatible sizes or if any element differ in a one-to-one comparison, `true` otherwise. If one of the operand is a scalar, the operator will be applied individually to all elements of the matrix.

mat .. mat2

mat .. imat

imat .. mat

Return a *matrix* resulting from the row-oriented (horizontal) concatenation of the left and right operands. If the first element of the right operand `mat` (third case) is an integer, the resulting matrix will be a *imatrix* instead.

mat .. cmat

imat .. cmat

cmat .. mat

cmat .. imat

cmat .. cmat2

Return a *cmatrix* resulting from the row-oriented (horizontal) concatenation of the left and right operands.

imat .. imat2

Return a *imatrix* resulting from the row-oriented (horizontal) concatenation of the left and right operands.

7 Iterators

ipairs(*mat*)

Return an *ipairs* iterator suitable for generic `for` loops. The returned values are those given by `mat[i]`.

8 C API

This C Application Programming Interface describes only the C functions declared in the scripting language and used by the higher level functions and methods presented before in this chapter. For more functions and details, see the C headers. The `const` vectors and matrices are inputs, while the non-`const` vectors and matrices are outpouts or are modified *inplace*.

8.1 Vector

void **mad_vec_fill**(*num_t* x, *num_t* r[], *ssz_t* n)

void **mad_cvec_fill**(*cpx_t* x, *cpx_t* r[], *ssz_t* n)

void **mad_ivec_fill**(*idx_t* x, *idx_t* r[], *ssz_t* n)

Return the vector `r` of size `n` filled with the value of `x`.

void **mad_vec_roll**(*num_t* x[], *ssz_t* n, int nroll)

void **mad_cvec_roll**(*cpx_t* x[], *ssz_t* n, int nroll)

void **mad_ivec_roll**(*idx_t* x[], *ssz_t* n, int nroll)

Roll in place the values of the elements of the vector `x` of size `n` by `nroll`.

void **mad_vec_copy**(const *num_t* x[], *num_t* r[], *ssz_t* n)

void **mad_vec_copyv**(const *num_t* x[], *cpx_t* r[], *ssz_t* n)

void **mad_cvec_copy**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* n)

void **mad_ivec_copy**(const *idx_t* x[], *idx_t* r[], *ssz_t* n)

Fill the vector `r` of size `n` with the content of the vector `x`.

void **mad_vec_minmax**(const *num_t* x[], *log_t* absf, *idx_t* r[2], *ssz_t* n)

void **mad_cvec_minmax**(const *cpx_t* x[], *idx_t* r[2], *ssz_t* n)

void **mad_ivec_minmax**(const *idx_t* x[], *log_t* absf, *idx_t* r[2], *ssz_t* n)

Return in `r` the indexes of the minimum and maximum values of the elements of the vector `x` of size `n`. If `absf = TRUE`, the function `abs()` is applied to the elements before comparison.

num_t **mad_vec_eval**(const *num_t* x[], *num_t* x0, *ssz_t* n)

void **mad_cvec_eval_r**(const *cpx_t* x[], *num_t* x0_re, *num_t* x0_im, *cpx_t* *r, *ssz_t* n)

Return in `r` or directly the evaluation of the vector `x` of size `n` at the point `x0` using Honer's scheme.

num_t **mad_vec_sum**(const *num_t* x[], *ssz_t* n)

void **mad_cvec_sum_r**(const *cpx_t* x[], *cpx_t* *r, *ssz_t* n)

num_t **mad_vec_ksum**(const *num_t* x[], *ssz_t* n)

void **mad_cvec_ksum_r**(const *cpx_t* x[], *cpx_t* *r, *ssz_t* n)

Return in `r` or directly the sum of the values of the elements of the vector `x` of size `n`. The *k* versions use the Neumaier variants of the Kahan sum.

num_t **mad_vec_mean**(const *num_t* x[], *ssz_t* n)

void **mad_cvec_mean_r**(const *cpx_t* x[], *cpx_t* *r, *ssz_t* n)

Return in `r` or directly the mean of the vector `x` of size `n`.

num_t **mad_vec_var**(const *num_t* x[], *ssz_t* n)

void **mad_cvec_var_r**(const *cpx_t* x[], *cpx_t* *r, *ssz_t* n)

Return in `r` or directly the unbiased variance with 2nd order correction of the vector `x` of size `n`.

void **mad_vec_center**(const *num_t* x[], *num_t* r[], *ssz_t* n)

void **mad_cvec_center**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* n)

Return in `r` the centered, vector `x` of size `n` equivalent to `x[i] - mean(x)`.

num_t **mad_vec_norm**(const *num_t* x[], *ssz_t* n)

num_t **mad_cvec_norm**(const *cpx_t* x[], *ssz_t* n)

Return the norm of the vector `x` of size `n`.

num_t **mad_vec_dist**(const *num_t* x[], const *num_t* y[], *ssz_t* n)

num_t **mad_vec_distv**(const *num_t* x[], const *cpx_t* y[], *ssz_t* n)

num_t **mad_cvec_dist**(const *cpx_t* x[], const *cpx_t* y[], *ssz_t* n)

num_t **mad_cvec_distv**(const *cpx_t* x[], const *num_t* y[], *ssz_t* n)

Return the distance between the vectors `x` and `y` of size `n` equivalent to `norm(x - y)`.

num_t **mad_vec_dot**(const *num_t* x[], const *num_t* y[], *ssz_t* n)

void **mad_cvec_dot_r**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* *r, *ssz_t* n)

void **mad_cvec_dotv_r**(const *cpx_t* x[], const *num_t* y[], *cpx_t* *r, *ssz_t* n)

num_t **mad_vec_kdot**(const *num_t* x[], const *num_t* y[], *ssz_t* n)

void **mad_cvec_kdot_r**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* *r, *ssz_t* n)

void **mad_cvec_kdotv_r**(const *cpx_t* x[], const *num_t* y[], *cpx_t* *r, *ssz_t* n)

Return in `r` or directly the dot product between the vectors `x` and `y` of size `n`. The *k* versions use the Neumaier variants of the Kahan sum.

void **mad_vec_cplx**(const *num_t* re_[], const *num_t* im_[], *cpx_t* r[], *ssz_t* n)

Convert the real and imaginary vectors `re` and `im` of size `n` into the complex vector `r`.

void **mad_cvec_reim**(const *cpx_t* x[], *num_t* re_[], *num_t* ri_[], *ssz_t* n)

Split the complex vector `x` of size `n` into the real vector `re` and the imaginary vector `ri`.

void **mad_cvec_conj**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* n)

Return in `r` the conjugate of the complex vector `x` of size `n`.

void **mad_vec_abs**(const *num_t* x[], *num_t* r[], *ssz_t* n)

void **mad_cvec_abs**(const *cpx_t* x[], *num_t* r[], *ssz_t* n)

Return in `r` the absolute value of the vector `x` of size `n`.

void **mad_vec_add**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* n)

void **mad_vec_addn**(const *num_t* x[], *num_t* y, *num_t* r[], *ssz_t* n)

void **mad_vec_addc_r**(const *num_t* x[], *num_t* y_re, *num_t* y_im, *cpx_t* r[], *ssz_t* n)

void **mad_cvec_add**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)

void **mad_cvec_addv**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_addn**(const *cpx_t* x[], *num_t* y, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_addc_r**(const *cpx_t* x[], *num_t* y_re, *num_t* y_im, *cpx_t* r[], *ssz_t* n)
void **mad_ivec_add**(const *idx_t* x[], const *idx_t* y[], *idx_t* r[], *ssz_t* n)
void **mad_ivec_addn**(const *idx_t* x[], *idx_t* y, *idx_t* r[], *ssz_t* n)

Return in `r` the sum of the scalar or vectors `x` and `y` of size `n`.

void **mad_vec_sub**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* n)
void **mad_vec_subv**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_vec_subn**(const *num_t* y[], *num_t* x, *num_t* r[], *ssz_t* n)
void **mad_vec_subc_r**(const *num_t* y[], *num_t* x_re, *num_t* x_im, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_sub**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_subv**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_subn**(const *cpx_t* y[], *num_t* x, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_subc_r**(const *cpx_t* y[], *num_t* x_re, *num_t* x_im, *cpx_t* r[], *ssz_t* n)
void **mad_ivec_sub**(const *idx_t* x[], const *idx_t* y[], *idx_t* r[], *ssz_t* n)
void **mad_ivec_subn**(const *idx_t* y[], *idx_t* x, *idx_t* r[], *ssz_t* n)

Return in `r` the difference between the scalar or vectors `x` and `y` of size `n`.

void **mad_vec_mul**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* n)
void **mad_vec_muln**(const *num_t* x[], *num_t* y, *num_t* r[], *ssz_t* n)
void **mad_vec_mulc_r**(const *num_t* x[], *num_t* y_re, *num_t* y_im, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_mul**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_mulv**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_muln**(const *cpx_t* x[], *num_t* y, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_mulc_r**(const *cpx_t* x[], *num_t* y_re, *num_t* y_im, *cpx_t* r[], *ssz_t* n)
void **mad_ivec_mul**(const *idx_t* x[], const *idx_t* y[], *idx_t* r[], *ssz_t* n)
void **mad_ivec_muln**(const *idx_t* x[], *idx_t* y, *idx_t* r[], *ssz_t* n)

Return in `r` the product of the scalar or vectors `x` and `y` of size `n`.

void **mad_vec_div**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* n)
void **mad_vec_divv**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_vec_divn**(const *num_t* y[], *num_t* x, *num_t* r[], *ssz_t* n)
void **mad_vec_divc_r**(const *num_t* y[], *num_t* x_re, *num_t* x_im, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_div**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_divv**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* n)
void **mad_cvec_divn**(const *cpx_t* y[], *num_t* x, *cpx_t* r[], *ssz_t* n)
void **mad_cvec_divc_r**(const *cpx_t* y[], *num_t* x_re, *num_t* x_im, *cpx_t* r[], *ssz_t* n)
void **mad_ivec_divn**(const *idx_t* x[], *idx_t* y, *idx_t* r[], *ssz_t* n)

Return in `r` the division of the scalar or vectors `x` and `y` of size `n`.

void **mad_ivec_modn**(const *idx_t* x[], *idx_t* y, *idx_t* r[], *ssz_t* n)

Return in `r` the modulo of the integer vector `x` of size `n` by the integer `y`.

void **mad_vec_kadd**(int k, const *num_t* a[], const *num_t* *x[], *num_t* r[], *ssz_t* n)

void **mad_cvec_kadd**(int k, const *cpx_t* a[], const *cpx_t* *x[], *cpx_t* r[], *ssz_t* n)

Return in `r` the linear combination of the `k` vectors in `x` of size `n` scaled by the `k` scalars in `a`.

void **mad_vec_fft**(const *num_t* x[], *cpx_t* r[], *ssz_t* n)

void **mad_cvec_fft**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* n)

void **mad_cvec_ifft**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* n)

Return in the vector `r` the 1D FFT and inverse of the vector `x` of size `n`.

void **mad_vec_rfft**(const *num_t* x[], *cpx_t* r[], *ssz_t* n)

Return in the vector `r` of size `n/2+1` the 1D *real* FFT of the vector `x` of size `n`.

void **mad_cvec_irfft**(const *cpx_t* x[], *num_t* r[], *ssz_t* n)

Return in the vector `r` of size `n` the 1D *real* FFT inverse of the vector `x` of size `n/2+1`.

void **mad_vec_nfft**(const *num_t* x[], const *num_t* x_node[], *cpx_t* r[], *ssz_t* n, *ssz_t* nr)

void **mad_cvec_nfft**(const *cpx_t* x[], const *num_t* x_node[], *cpx_t* r[], *ssz_t* n, *ssz_t* nr)

Return in the vector `r` of size `nr` the 1D non-equispaced FFT of the vectors `x` and `x_node` of size `n`.

void **mad_cvec_infft**(const *cpx_t* x[], const *num_t* r_node[], *cpx_t* r[], *ssz_t* n, *ssz_t* nx)

Return in the vector `r` of size `n` the 1D non-equispaced FFT inverse of the vector `x` of size `nx` and the vector `r_node` of size `n`. Note that `r_node` here is the same vector as `x_node` in the 1D non-equispaced forward FFT.

8.2 Matrix

void **mad_mat_rev**(*num_t* x[], *ssz_t* m, *ssz_t* n, int d)

void **mad_cmat_rev**(*cpx_t* x[], *ssz_t* m, *ssz_t* n, int d)

void **mad_imat_rev**(*idx_t* x[], *ssz_t* m, *ssz_t* n, int d)

Reverse in place the matrix `x` following the direction `d in {0,1,2,3}` for respectively the entire matrix, each row, each column and the diagonal.

void **mad_mat_center**(*num_t* x[], *ssz_t* m, *ssz_t* n, int d)

void **mad_cmat_center**(*cpx_t* x[], *ssz_t* m, *ssz_t* n, int d)

Center in place the matrix `x` following the direction `d in {0,1,2,3}` for respectively the entire matrix, each row, each column and the diagonal.

void **mad_mat_roll**(*num_t* x[], *ssz_t* m, *ssz_t* n, int mroll, int nroll)

void **mad_cmat_roll**(*cpx_t* x[], *ssz_t* m, *ssz_t* n, int mroll, int nroll)

void **mad_imat_roll**(*idx_t* x[], *ssz_t* m, *ssz_t* n, int mroll, int nroll)

Roll in place the values of the elements of the matrix `x` of sizes `[m, n]` by `mroll` and `nroll`.

void **mad_mat_eye**(*num_t* x[], *num_t* v, *ssz_t* m, *ssz_t* n, *ssz_t* ldr)

void **mad_cmat_eye_r**(*cpx_t* x[], *num_t* v_re, *num_t* v_im, *ssz_t* m, *ssz_t* n, *ssz_t* ldr)

void **mad_imat_eye**(*idx_t* x[], *idx_t* v, *ssz_t* m, *ssz_t* n, *ssz_t* ldr)

Fill in place the matrix `x` of sizes `[m, n]` with zeros and `v` on the diagonal.

void **mad_mat_copy**(const *num_t* x[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* ldx, *ssz_t* ldr)

void **mad_mat_copym**(const *num_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* ldx, *ssz_t* ldr)

void **mad_cmat_copy**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* ldx, *ssz_t* ldr)

void **mad_imat_copy**(const *idx_t* x[], *idx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* ldx, *ssz_t* ldr)

void **mad_imat_copym**(const *idx_t* x[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* ldx, *ssz_t* ldr)

Fill the matrix `r` of sizes `[m, n]` and leading dimension `ldr` with the content of the matrix `x` of sizes `[m, n]` and leading dimension `ldx`.

void **mad_mat_trans**(const *num_t* x[], *num_t* r[], *ssz_t* m, *ssz_t* n)

void **mad_cmat_trans**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n)

void **mad_cmat_ctrans**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n)

void **mad_imat_trans**(const *idx_t* x[], *idx_t* r[], *ssz_t* m, *ssz_t* n)

Fill the matrix `r` of sizes `[n, m]` with the (conjugate) transpose of the matrix `x` of sizes `[m, n]`.

void **mad_mat_mul**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_mat_mulm**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_mul**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_mulm**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

Fill the matrix `r` of sizes `[m, n]` with the product of the matrix `x` of sizes `[m, p]` by the matrix `y` of sizes `[p, n]`.

void **mad_mat_tmul**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_mat_tmulm**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_tmul**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_tmulm**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

Fill the matrix `r` of sizes `[m, n]` with the product of the transposed matrix `x` of sizes `[p, m]` by the matrix `y` of sizes `[p, n]`.

void **mad_mat_mult**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_mat_multm**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_mult**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_multm**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

Fill the matrix `r` of sizes `[m, n]` with the product of the matrix `x` of sizes `[m, p]` and the transposed matrix `y` of sizes `[n, p]`.

void **mad_mat_dmul**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_mat_dmulm**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_dmul**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_dmulm**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

Fill the matrix `r` of size `[m, n]` with the product of the diagonal of the matrix `x` of sizes `[m, p]` by the matrix `y` of sizes `[p, n]`. If `p = 1` then `x` will be interpreted as the diagonal of a square matrix.

void **mad_mat_muld**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_mat_muldm**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_muld**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

void **mad_cmat_muldm**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

Fill the matrix `r` of sizes `[m, n]` with the product of the matrix `x` of sizes `[m, p]` by the diagonal of the matrix `y` of sizes `[p, n]`. If `p = 1` then `y` will be interpreted as the diagonal of a square matrix.

int **mad_mat_div**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond)

int **mad_mat_divm**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond)

int **mad_cmat_div**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond)

int **mad_cmat_divm**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond)

Fill the matrix `r` of sizes `[m, n]` with the division of the matrx `x` of sizes `[m, p]` by the matrix `y` of sizes `[n, p]`. The conditional number `rcond` is used by the solver to determine the effective rank of non-square systems. It returns the rank of the system.

int **mad_mat_invn**(const *num_t* y[], *num_t* x, *num_t* r[], *ssz_t* m, *ssz_t* n, *num_t* rcond)

int **mad_mat_invc_r**(const *num_t* y[], *num_t* x_re, *num_t* x_im, *cpx_t* r[], *ssz_t* m, *ssz_t* n, *num_t* rcond)

int **mad_cmat_invn**(const *cpx_t* y[], *num_t* x, *cpx_t* r[], *ssz_t* m, *ssz_t* n, *num_t* rcond)

int **mad_cmat_invc_r**(const *cpx_t* y[], *num_t* x_re, *num_t* x_im, *cpx_t* r[], *ssz_t* m, *ssz_t* n, *num_t* rcond)

Fill the matrix `r` of sizes `[n, m]` with the inverse of the matrix `y` of sizes `[m, n]` scaled by the scalar `x`. The conditional number `rcond` is used by the solver to determine the effective rank of non-square systems. It returns the rank of the system.

int **mad_mat_solve**(const *num_t* a[], const *num_t* b[], *num_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond)

int **mad_cmat_solve**(const *cpx_t* a[], const *cpx_t* b[], *cpx_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond)

Fill the matrix `x` of sizes `[n, p]` with the minimum-norm solution of the linear least square problem $\min \|Ax - B\|$ where A is the matrix `a` of sizes `[m, n]` and B is the matrix `b` of sizes `[m, p]`, using LU, QR or LQ factorisation depending on the shape of the system. The conditional number `rcond` is used by the solver to determine the effective rank of non-square system. It returns the rank of the system.

int **mad_mat_ssolve**(const *num_t* a[], const *num_t* b[], *num_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond, *num_t* s_[])

int **mad_cmat_ssolve**(const *cpx_t* a[], const *cpx_t* b[], *cpx_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* rcond, *num_t* s_[])

Fill in the matrix `x` of sizes `[n, p]` with the minimum-norm solution of the linear least square problem $\min \|Ax - B\|$ where A is the matrix `a` of sizes `[m, n]` and B is the matrix `b` of sizes `[m, p]`, using SVD factorisation. The conditional number `rcond` is used by the solver to determine the effective rank of non-square system. It returns the rank of the system and fill the optional column vector `s` of size `min(m,n)` with the singular values.

int **mad_mat_gsolve**(const *num_t* a[], const *num_t* b[], const *num_t* c[], const *num_t* d[], *num_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* *nrm_)

int **mad_cmat_gsolve**(const *cpx_t* a[], const *cpx_t* b[], const *cpx_t* c[], const *cpx_t* d[], *cpx_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* p, *num_t* *nrm_)

Fill the column vector `x` of size `n` with the minimum-norm solution of the linear least square problem $\min \|Ax - C\|$ under the constraint $Bx = D$ where A is a matrix `a` of sizes `[m, n]`, B is a matrix

b of sizes [p, n], C is a column vector of size m and D is a column vector of size p, using QR or LQ factorisation depending on the shape of the system. This function also returns the status info and optionally the norm of the residues in the nrm.

int **mad_mat_gmsolve**(const *num_t* a[], const *num_t* b[], const *num_t* d[], *num_t* x[], *num_t* y[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

int **mad_cmat_gmsolve**(const *cpx_t* a[], const *cpx_t* b[], const *cpx_t* d[], *cpx_t* x[], *cpx_t* y[], *ssz_t* m, *ssz_t* n, *ssz_t* p)

Fill the column vector x of size n and column vector y of size p with the minimum-norm solution of the linear Gauss-Markov problem $\min_x \|y\|$ under the constraint $Ax + By = D$ where A is a matrix a of sizes [m, n], B is a matrix b of sizes [m, p], and D is a column vector of size m, using QR or LQ factorisation depending on the shape of the system. This function also returns the status info.

int **mad_mat_nsolve**(const *num_t* a[], const *num_t* b[], *num_t* x[], *ssz_t* m, *ssz_t* n, *ssz_t* nc, *num_t* rcond, *num_t* r_[])

Fill the column vector x (of correctors kicks) of size n with the minimum-norm solution of the linear (best-kick) least square problem $\min \|Ax - B\|$ where A is the (response) matrix a of sizes [m, n] and B is a column vector (of monitors readings) of size m, using the MICADO[Page 232, 3] algorithm based on the Householder-Golub method [MICADO]. The argument nc is the maximum number of correctors to use with $0 < n_c \leq n$ and the argument tol is a convergence threshold (on the residues) to stop the (orbit) correction if $\|Ax - B\| \leq m \times$ tol. This function also returns the updated number of correctors nc effectively used during the correction and the residues in the optional column vector r of size m.

int **mad_mat_pcacnd**(const *num_t* a[], *idx_t* ic[], *ssz_t* m, *ssz_t* n, *ssz_t* ns, *num_t* cut, *num_t* s_[])

int **mad_cmat_pcacnd**(const *cpx_t* a[], *idx_t* ic[], *ssz_t* m, *ssz_t* n, *ssz_t* ns, *num_t* cut, *num_t* s_[])

Fill the column vector ic of size n with the indexes of the columns to remove from the matrix a of sizes [m, n] using the Principal Component Analysis. The argument ns is the maximum number of singular values to consider and rcond is the conditioning number used to select the singular values versus the largest one, i.e. consider the ns larger singular values σ_i such that $\sigma_i > \sigma_{\max} \times$ rcond. This function also returns the column vector of size min(m,n) filled with the singluar values. Default: ns_ = ncol, rcond_ = eps.

int **mad_mat_svdcnd**(const *num_t* a[], *idx_t* ic[], *ssz_t* m, *ssz_t* n, *ssz_t* ns, *num_t* cut, *num_t* s_[], *num_t* tol)

int **mad_cmat_svdcnd**(const *cpx_t* a[], *idx_t* ic[], *ssz_t* m, *ssz_t* n, *ssz_t* ns, *num_t* cut, *num_t* s_[], *num_t* tol)

Fill the column vector ic of size n with the indexes of the columns to remove from the matrix a of sizes [m, n] based on the analysis of the right matrix V from the SVD decomposition USV. The argument ns is the maximum number of singular values to consider and rcond is the conditioning number used to select the singular values versus the largest one, i.e. consider the ns larger singular values σ_i such that $\sigma_i > \sigma_{\max} \times$ rcond. The argument tol is a threshold similar to rcond used to reject components in V that have similar or opposite effect than components already encountered. This function also returns the real column vector of size min(m,n) filled with the singluar values. Default: ns_ = min(m,n), rcond_ = eps.

int **mad_mat_svd**(const *num_t* x[], *num_t* u[], *num_t* s[], *num_t* v[], *ssz_t* m, *ssz_t* n)

int **mad_cmat_svd**(const *cpx_t* x[], *cpx_t* u[], *num_t* s[], *cpx_t* v[], *ssz_t* m, *ssz_t* n)

Fill the column vector `s` of size `min(m,n)` with the singular values, and the two matrices `u` of sizes `[m, m]` and `v` of sizes `[n, n]` with the SVD factorisation of the matrix `x` of sizes `[m,n]`, and returns the status `info`. The singular values are positive and sorted in decreasing order of values, i.e. largest first.

int **mad_mat_eigen**(const *num_t* x[], *cpx_t* w[], *num_t* vl[], *num_t* vr[], *ssz_t* n)

int **mad_cmat_eigen**(const *cpx_t* x[], *cpx_t* w[], *cpx_t* vl[], *cpx_t* vr[], *ssz_t* n)

Fill the column vector `w` of size `n` with the eigenvalues followed by the status `info` and the two optional matrices `vr` and `vl` of sizes `[n, n]` containing the left and right eigenvectors resulting from the Eigen Decomposition of the square matrix `x` of sizes `[n, n]`. The eigenvectors are normalized to have unit Euclidean norm and their largest component real, and satisfy $Xv_r = \lambda v_r$ and $v_l X = \lambda v_l$.

int **mad_mat_det**(const *num_t* x[], *num_t* *r, *ssz_t* n)

int **mad_cmat_det**(const *cpx_t* x[], *cpx_t* *r, *ssz_t* n)

Return in `r`, the Determinant of the square matrix `mat` of sizes `[n, n]` using LU factorisation for better numerical stability, and return the status `info`.

void **mad_mat_fft**(const *num_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n)

void **mad_cmat_fft**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n)

void **mad_cmat_ifft**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n)

Fill the matrix `r` with the 2D FFT and inverse of the matrix `x` of sizes `[m, n]`.

void **mad_mat_rfft**(const *num_t* x[], *cpx_t* r[], *ssz_t* m, *ssz_t* n)

Fill the matrix `r` of sizes `[m, n/2+1]` with the 2D *real* FFT of the matrix `x` of sizes `[m, n]`.

void **mad_cmat_irfft**(const *cpx_t* x[], *num_t* r[], *ssz_t* m, *ssz_t* n)

Fill the matrix `r` of sizes `[m, n]` with the 1D *real* FFT inverse of the matrix `x` of sizes `[m, n/2+1]`.

void **mad_mat_nfft**(const *num_t* x[], const *num_t* x_node[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* nr)

void **mad_cmat_nfft**(const *cpx_t* x[], const *num_t* x_node[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* nr)

Fill the matrix `r` of sizes `[m, nr]` with the 2D non-equispaced FFT of the matrices `x` and `x_node` of sizes `[m, n]`.

void **mad_cmat_infft**(const *cpx_t* x[], const *num_t* r_node[], *cpx_t* r[], *ssz_t* m, *ssz_t* n, *ssz_t* nx)

Fill the matrix `r` of sizes `[m, n]` with the 2D non-equispaced FFT inverse of the matrix `x` of sizes `[m, nx]` and the matrix `r_node` of sizes `[m, n]`. Note that `r_node` here is the same matrix as `x_node` in the 2D non-equispaced forward FFT.

void **mad_mat_sympconj**(const *num_t* x[], *num_t* r[], *ssz_t* n)

void **mad_cmat_sympconj**(const *cpx_t* x[], *cpx_t* r[], *ssz_t* n)

Return in `r` the symplectic 'conjugate' of the vector `x` of size `n`.

num_t **mad_mat_symperr**(const *num_t* x[], *num_t* r_[], *ssz_t* n, *num_t* *tol_)

num_t **mad_cmat_symperr**(const *cpx_t* x[], *cpx_t* r_[], *ssz_t* n, *num_t* *tol_)

Return the norm of the symplectic error and fill the optional matrix `r` with the symplectic deviation of the matrix `x`. The optional argument `tol` is used as the tolerance to check if the matrix `x` is symplectic or not, and saves the result as `0` (non-symplectic) or `1` (symplectic) within tol for output.

void **mad_vec_dif**(const *num_t* x[], const *num_t* y[], *num_t* r[], *ssz_t* n)

void **mad_vec_difv**(const *num_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)

void **mad_cvec_dif**(const *cpx_t* x[], const *cpx_t* y[], *cpx_t* r[], *ssz_t* n)

void **mad_cvec_difv**(const *cpx_t* x[], const *num_t* y[], *cpx_t* r[], *ssz_t* n)

Fill the matrix `r` of sizes `[m, n]` with the absolute or relative differences between the elements of the matrix `x` and `y` with compatible sizes. The relative difference is taken for the values with magnitude greater than 1, otherwise it takes the absolute difference.

8.3 Rotations

void **mad_mat_rot**(*num_t* x[2 * 2], *num_t* a)

Fill the matrix `x` with a 2D rotation of angle `a`.

void **mad_mat_rotx**(*num_t* x[3 * 3], *num_t* ax)

void **mad_mat_roty**(*num_t* x[3 * 3], *num_t* ay)

void **mad_mat_rotz**(*num_t* x[3 * 3], *num_t* az)

Fill the matrix `x` with the 3D rotation of angle `a?` around the axis given by the suffix `? in {x,y,z}`.

void **mad_mat_rotxy**(*num_t* x[3 * 3], *num_t* ax, *num_t* ay, *log_t* inv)

void **mad_mat_rotxz**(*num_t* x[3 * 3], *num_t* ax, *num_t* az, *log_t* inv)

void **mad_mat_rotyz**(*num_t* x[3 * 3], *num_t* ay, *num_t* az, *log_t* inv)

Fill the matrix `x` with the two successive 3D rotations of angles `a?` around the two axis given by the suffixes `? in {x,y,z}`. If `inv = 1` returns the inverse rotations, i.e. the transpose of the matrix `x`. Note that the first rotation changes the axis orientation of the second rotation.

void **mad_mat_rotxyz**(*num_t* x[3 * 3], *num_t* ax, *num_t* ay, *num_t* az, *log_t* inv)

void **mad_mat_rotxzy**(*num_t* x[3 * 3], *num_t* ax, *num_t* ay, *num_t* az, *log_t* inv)

void **mad_mat_rotyxz**(*num_t* x[3 * 3], *num_t* ax, *num_t* ay, *num_t* az, *log_t* inv)

Fill the matrix `x` with the three successive 3D rotations of angles `a?` around the three axis given by the suffixes `? in {x,y,z}`. If `inv = 1` returns the inverse rotations, i.e. the transpose of the matrix `x`. Note that the first rotation changes the axis orientation of the second rotation, which changes the axis orientation of the third rotation.

void **mad_mat_torotxyz**(const *num_t* x[3 * 3], *num_t* r[3], *log_t* inv)

void **mad_mat_torotxzy**(const *num_t* x[3 * 3], *num_t* r[3], *log_t* inv)

void **mad_mat_torotyxz**(const *num_t* x[3 * 3], *num_t* r[3], *log_t* inv)

Fill the vector of the three angles `r` around the axis `{x,y,z}`, `{x,z,y}` and `{y,x,z}` from the matrix `x`. If `inv = 1`, it takes the inverse rotations, i.e. the transpose of the matrix `x`.

void **mad_mat_rotv**(*num_t* x[3 * 3], const *num_t* v[3], *num_t* a, *log_t* inv)

Fill the matrix `x` with the 3D rotation of angle `a` around the vector `v`. If `inv = 1` returns the inverse rotations, i.e. the transpose of the matrix.

num_t **mad_mat_torotv**(const *num_t* x[3 * 3], *num_t* v_[3], *log_t* inv)

Return the angle and fill the optional vector `v` with the 3D rotations in `x`. If `inv = 1`, it takes the inverse rotations, i.e. the transpose of the matrix `x`.

void **mad_mat_rotq**(*num_t* x[3 * 3], const *num_t* q[4], *log_t* inv)

Fill the matrix `x` with the 3D rotation given by the quaternion `q`. If `inv = 1` returns the inverse rotations, i.e. the transpose of the matrix.

void **mad_mat_torotq**(const *num_t* x[3 * 3], *num_t* q[4], *log_t* inv)

Fill the quaternion `q` with the 3D rotations in `x`. If `inv = 1`, it takes the inverse rotations, i.e. the transpose of the matrix `x`.

8.4 Misalignments

void **mad_mat_rtbar**(*num_t* Rb[3 * 3], *num_t* Tb[3], *num_t* el, *num_t* ang, *num_t* tlt, const *num_t* R_[3 * 3], const *num_t* T[3])

Compute as output the rotation matrix `Rb`, i.e. $\bar{R}$, and the translation vector `Tb`, i.e. $\bar{T}$, used to restore the global frame at exit of a misaligned element in survey, given as input the element length `el`, angle `ang`, tilt `tlt`, and the rotation matrix `R` and the translation vector `T` at entry.

8.5 Miscellaneous

void **mad_fft_cleanup**(void)

Cleanup data allocated by the FFTW library.

9 References

Chapter 37. Differential Algebra

This chapter describes real *tpsa* and complex *ctpsa* objects as supported by MAD-NG. The module for the Generalized Truncated Power Series Algebra (GTPSA) that represents parametric multivariate truncated Taylor series is not exposed, only the contructors are visible from the `MAD` environment and thus, TPSAs are handled directly by their methods or by the generic functions of the same name from the module `MAD.gmath`. Note that both *tpsa* and *ctpsa* are defined as C structure for direct compliance with the C API.

1 Introduction

TPSAs are numerical objects representing n-th degrees Taylor polynomial approximation of some functions $f(x)$ about $x = a$. They are a powerful differential algebra tool for solving physics problems described by differential equations and for perturbation theory, e.g. for solving motion equations, but also for estimating uncertainties, modelling multidimensional distributions or calculating multivariate derivatives for optimization. There are often misunderstandings about their accuracy and limitations, so it is useful to clarify here some of these aspects here.

To begin with, GTPSAs represent multivariate Taylor series truncated at order n, and thus behave like n-th degrees multivariate polynomials with coefficients in $\mathbb{R}$ or $\mathbb{C}$. MAD-NG supports GTPSAs with thousands of variables and/or parameters of arbitrary order each, up to a maximum total order of 63, but Taylor series with alternating signs in their coefficients can quickly be subject to numerical instabilities and catastrophic cancellation as orders increase.

Other methods are not better and suffer from the same problem and more, such as symbolic differentiation, which can lead to inefficient code due to the size of the analytical expressions, or numerical differentiation, which can introduce round-off errors in the discretisation process and cancellation. Both classical methods are more problematic when computing high order derivatives, where complexity and errors increase.

1.1 Representation

A TPSA in the variable x at order n in the neighbourhood of the point a in the domain of the function f, noted $T_f^n(x;a)$, has the following representation:

$$\begin{aligned} T_f^n(x;a) &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n \\ &= \sum_{k=0}^{n} \frac{f_a^{(k)}}{k!}(x-a)^k \end{aligned}$$

where the terms $\frac{f_a^{(k)}}{k!}$ are the coefficients stored in the *tpsa* and *ctpsa* objects.

The calculation of these coefficients uses a technique known as automatic differentiation (AD) which operates as polynomials over the augmented (differential) algebra of dual number, *without any approximation*, being exact to numerical precision.

The validity of the polynomial representation $T_f^n(x;a)$ for the real or complex analytic function f is characterized by the convergence of the remainder when the order n goes to infinity:

$$\lim_{n\to\infty} R_f^n(x;a) = \lim_{n\to\infty} f_a(x) - T_f^n(x;a) = 0$$

and the radius of convergence h of $T_f^n(x;a)$ nearby the point a is given by:

$$\min_{h>0} \lim_{n\to\infty} R_f^n(x \pm h;a) \neq 0.$$

By using the mean value theorem recursively we can derive the explicit mean-value form of the remainder:

$$R_f^n(x;a) = \frac{f_a^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}$$

for some ξ strictly between x and a, leading to the mean-value form of the Taylor's theorem:

$$f_a(x) = T_f^n(x;a) + R_f^n(x;a) = \sum_{k=0}^{n} \frac{f_a^{(k)}}{k!}(x-a)^k + \frac{f_a^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}$$

Note that a large radius of convergence does not necessarily mean rapid convergence of the Taylor series to the function, although there is a relationship between the rate of convergence, the function f, the point a and the length h. Nevertheless, Taylor series are known to be slow to converge in most cases for numerical applications, except in some cases where appropriate range reduction or convergence acceleration methods give good results. Thus, Taylor series should not be used as interpolation functions when better formulas exist for this purpose, see for example fixed-point or minmax algorithms.

In our practice, a truncation error is always present due to the truncated nature of the TPSA at order n, but it is rarely calculated analytically for complex systems as it can be estimated by comparing the calculations at high and low orders, and determining the lowest order for which the result is sufficiently stable.

By extension, a TPSA in the two variables x and y at order 2 in the neighbourhood of the point (a,b) in the domain of the function f, noted $T_f^2(x,y;a,b)$, has the following representation:

$$\begin{aligned} T_f^2(x,y;a,b) = f(a,b) + &\left(\left.\frac{\partial f}{\partial x}\right|_{(a,b)} (x-a) + \left.\frac{\partial f}{\partial y}\right|_{(a,b)} (y-b) \right) \\ &+ \frac{1}{2!}\left(\left.\frac{\partial^2 f}{\partial x^2}\right|_{(a,b)} (x-a)^2 + 2\left.\frac{\partial^2 f}{\partial x \partial y}\right|_{(a,b)} (x-a)(y-b) + \left.\frac{\partial^2 f}{\partial y^2}\right|_{(a,b)} (y-b)^2 \right) \end{aligned}$$

where the large brackets are grouping the terms in homogeneous polynomials, as stored in the *tpsa* and *ctpsa* objects. The central term of the second order $2\frac{\partial^2 f}{\partial x \partial y}$ emphasises the reason why the function f must be analytic and independent of the integration path as it implies $\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial^2 f}{\partial y \partial x}$ and stores the value (scaled by $\frac{1}{2}$) as the coefficient of the monomial $x^1 y^1$. This is an important consideration to keep in mind regarding TPSA, but it is not a pactical limitation due to the conservative nature of our applications described by Hamiltonian vector fields.

The generalization to a TPSA of ν variables X at order n nearby the point A in the ν-dimensional domain of the function f, noted $T_f^n(X;A)$, has the following representation:

$$T_f^n(X;A) = \sum_{k=0}^{n} \frac{f_A^{(k)}}{k!}(X;A)^k = \sum_{k=0}^{n} \frac{1}{k!} \sum_{|\vec{m}|=k} \binom{k}{\vec{m}} \left.\frac{\partial^k f}{\partial X^{\vec{m}}}\right|_A (X;A)^{\vec{m}}$$

where the term $\binom{k}{\vec{m}} = \frac{k!}{c_1!\,c_2!..c_\nu!}$ is the multinomial coefficient with $\vec{m}$ the vector of ν variables orders $c_i, i = 1..\nu$ in the monomial and $|\vec{m}| = \sum_i c_i$ its total order. Again, we may mention that each term $\frac{1}{k!}\binom{k}{\vec{m}}\frac{\partial^k f}{\partial X^{\vec{m}}}\Big|_A$ corresponds strictly to a coefficient stored in the *tpsa* and *ctpsa* objects.

An important point to mention is related to the *multinomial coefficient* and its relevance when computing physical quantities such as high order anharmonicities, e.g. chromaticities. When the physical quantity corresponds to the derivative of the function $f_A^{(k)}$, the coefficient must be multiplied by $c_1!\,c_2!\,..c_\nu!$ in order to obtain the correct value.

1.2 Approximation

As already said, TPSAs do not perform approximations for orders $0..n$ and the Taylor's theorem gives an explicit form of the remainder for the truncation error of higher orders, while all derivatives are computed using AD. AD relies on the fact that any computer program can execute a sequence of elementary arithmetic operations and functions, and apply the chain rule to them repeatedly to automatically compute the derivatives to machine precision.

So when TPSAs introduce appromixation errors? When they are used as *interpolation functions* to approximate by substitution or perturbation, values at positions $a + h$ away from their initial point a:

$$T_f^n(x+h;a) = \sum_{k=0}^{n} \frac{f_a^{(k)}}{k!}(x-a+h)^k \quad \neq \quad \sum_{k=0}^{n} \frac{f_{a+h}^{(k)}}{k!}(x-a-h)^k = T_f^n(x;a+h)$$

where the approximation error at order k is given by:

$$\left|f_{a+h}^{(k)} - f_a^{(k)}\right| \approx \frac{1}{|2h|}\left|\frac{\mathrm{d}^k T_f^n(x;a+h)}{\mathrm{d}x^k} - \frac{\mathrm{d}^k T_f^n(x+h;a)}{\mathrm{d}x^k}\right| + \mathcal{O}(k+1)$$

In summary, operations and functions on TPSAs are exact while TPSAs used as functions lead to approximations even within the radius of convergence, unlike infinite Taylor series. MAD-NG never uses TPSAs as interpolation functions, but of course the module does provide users with methods for interpolating functions.

1.3 Application

MAD-NG is a tracking code that never composes elements maps during tracking, but performs a *functional application* of elements physics to user-defined input differential maps modelled as sets of TPSAs (one per variable). Tracking particles orbits is a specific case where the "differential" maps are of order 0, i.e. they contain only the scalar part of the maps and no derivatives. Therefore, TPSAs must also behave as scalars in polymorphic codes like MAD-NG, so that the same equations of motion can be applied by the same functions to particle orbits and differential maps. Thus, the `track` command, and by extension the `cofind` (closed orbit search) and `twiss` commands, never use TPSAs as interpolation functions and the results are as accurate as for tracking particles orbits. In particular, it preserves the symplectic structure of the phase space if the applied elements maps are themselves symplectic maps.

Users may be tempted to compute or compose elements maps to model whole elements or even large lattice sections before applying them to some input differential maps in order to speed up tracking or parallelise

computations. But this approach leads to the two types of approximations that we have just explained: the resulting map is not only truncated, thus loosing local feed-down effects implied by e.g. a translation from orbit x to $x + h(s)$ along the path s or equivalently by the misalignment of the elements, but the derivatives are also approximated for each particle orbit by the global composition calculated on a nearby orbit, typically the zero orbit. So as the addition of floating point numbers is not associative, the composition of truncated maps is not associative too.

The following equations show the successive refinement of the type of calculations performed by the tracking codes, starting from the worst but common approximations at the top-left to the more general and accurate functional application without approximation at the bottom-right, as computed by MAD-NG:

$$\begin{aligned}(\mathcal{M}_n \circ \cdots \circ \mathcal{M}_2 \circ \mathcal{M}_1)(X_0) &\neq \mathcal{M}_n(\cdots(\mathcal{M}_2(\mathcal{M}_1(X_0)))\cdots) \\ &\neq \widetilde{\mathcal{M}}_n(\cdots(\widetilde{\mathcal{M}}_2(\widetilde{\mathcal{M}}_1(X_0)))\cdots) \\ &\neq \mathcal{F}_n(\cdots(\mathcal{F}_2(\mathcal{F}_1(X_0)))\cdots)\end{aligned}$$

where $\mathcal{M}_i$ is the i-th map computed at some *a priori* orbit (zero orbit), $\widetilde{\mathcal{M}}_i$ is the i-th map computed at the input orbit X_{i-1} which still implies some expansion, and finally $\mathcal{F}_i$ is the functional application of the full-fledged physics of the i-th map without any intermediate expansion, i.e. without calculating a differential map, and with all the required knownledge including the input orbit X_{i-1} to perform the exact calculation.

However, although MAD-NG only performs functional map applications (last right equation above) and never compute element maps or uses TPSAs as interpolation functions, it could be prone to small truncation errors during the computation of the non-linear normal forms which involves the composition of many orbitless maps, potentially breaking symplecticity of the resulting transformation for last order.

The modelling of multidimensional beam distributions is also possible with TPSAs, as when a linear phase space description is provided as initial conditions to the `twiss` command through, e.g. a `beta0` block. Extending the description of the initial phase space with high-order maps allows complex non-linear phase spaces to be modelled and their transformations along the lattice to be captured and analysed.

1.4 Performance

In principle, TPSAs should have equivalent performance to matrix/tensors for low orders and small number of variables, perhaps slightly slower at order 1 or 2 as the management of these data structures involves complex code and additional memory allocations. But from order 3 and higher, TPSA-based codes outperform matrix/tensor codes because the number of coefficients remains much smaller as shown in Fig. 37.1 and Fig. 37.2, and the complexity of the elementary operations (resp. multiplication) depends linearly (resp. quadratically) on the size of these data structures.

ν \ n	1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12	13
2	6	12	20	30	42	56	72	90	110	132	156	182
3	12	30	60	105	168	252	360	495	660	858	1092	1365
4	20	60	140	280	504	840	1320	1980	2860	4004	5460	7280
5	30	105	280	630	1260	2310	3960	6435	10010	15015	21840	30940
6	42	168	504	1260	2772	5544	10296	18018	30030	48048	74256	111384
7	56	252	840	2310	5544	12012	24024	45045	80080	136136	222768	352716
8	72	360	1320	3960	10296	24024	51480	102960	194480	350064	604656	1007760

Figure37.1: Number of coefficients in TPSAs for ν variables at order n is $\binom{n+\nu}{\nu} = \frac{(n+\nu)!}{n!\nu!}$.

ν \ n	1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	10	11	12	13
2	6	14	30	62	126	254	510	1022	2046	4094	8190	16382
3	12	39	120	363	1092	3279	9840	29523	88572	265719	797160	2391483
4	20	84	340	1364	5460	21844	87380	349524	1398100	5592404	22369620	89478484
5	30	155	780	3905	19530	97655	488280	2441405	12207030	61035155	305175780	1525878905
6	42	258	1554	9330	55986	335922	2015538	12093234	72559410	435356466	2612138802	15672832818
7	56	399	2800	19607	137256	960799	6725600	47079207	329554456	2306881199	16148168400	113037178807
8	72	584	4680	37448	299592	2396744	19173960	153391688	1227133512	9817068104	78536544840	628292358728

Figure37.2: Number of coefficients in tensors for ν variables at order n is $\sum_{k=0}^{n} \nu^{k+1} = \frac{\nu(\nu^{n+1}-1)}{\nu-1}$.

2 Constructors

3 Functions

4 Methods

5 Operators

6 Iterators

7 C API

Chapter 38. Differential Maps

This chapter describes real *damap* and complex *cdamap* objects as supported by MAD-NG. They are useful abstractions to represent non-linear parametric multivariate differential maps, i.e. Diffeomorphisms, Vector Fields, Exponential Maps and Lie Derivative. The module for the differential maps is not exposed, only the contructors are visible from the `MAD` environment and thus, differential maps are handled directly by their methods or by the generic functions of the same name from the module `MAD.gmath`. Note that *damap* and *cdamap* are defined as C structure for direct compliance with the C API.

Chapter 39. Miscellaneous Functions

This chapter lists some useful functions from the module `MAD.utility` that are complementary to the standard library for manipulating files, strings, tables, and more.

1 Files Functions

openfile(*filename_*, *mode_*, *extension_*)

filexists(*filename*)

fileisnewer(*filename1*, *filename2*, *timeattr_*)

filesplitname(*filename*)

mockfile

2 Formating Functions

printf(*str*, *...*)

fprintf(*file*, *str*, *...*)

assertf(*str*, *...*)

errorf(*str*, *...*)

3 Strings Functions

strinter(*str*, *var*, *policy_*)

strtrim(*str*, *ini_*)

strnum(*str*, *ini_*)

strident(*str*, *ini_*)

strquote(*str*, *ini_*)

strbracket(*str*, *ini_*)

strsplit(*str*, *seps*, *ini_*)

strqsplit(*str*, *seps*, *ini_*)

strqsplitall(*str*, *seps*, *ini_*, *r_*)

is_identifier(*str*)

4 Tables Functions

kpairs(*tbl*, *n_*)

tblrep(*val*, *n_*, *tbldst_*)

tblicpy(*tblsrc*, *mtflag_*, *tbldst_*)

tblcpy(*tblsrc*, *mtflag_*, *tbldst_*)

tbldeepcpy(*tblsrc*, *mtflag_*, *xrefs_*, *tbldst_*)

tblcat(*tblsrc1*, *tblsrc2*, *mtflag_*, *tbldst_*)

tblorder(*tbl*, *key*, *n_*)

5 Iterable Functions

rep(*x*, *n_*)

clearidxs(*a*, *i_*, *j_*)

setidxs(*a*, *k_*, *i_*, *j_*)

bsearch(*tbl*, *val*, *[cmp_,] low_*, *high_*)

lsearch(*tbl*, *val*, *[cmp_,] low_*, *high_*)

monotonic(*tbl*, *[strict_,] [cmp_,] low_*, *high_*)

6 Mappable Functions

clearkeys(*a*, *pred_*)

setkeys(*a*, *k_*, *i_*, *j_*)

countkeys(*a*)

keyscount(*a*, *c_*)

val2keys(*a*)

7 Conversion Functions

log2num(*log*)

num2log(*num*)

num2str(*num*)

int2str(*int*)

str2str(*str*)

str2cmp(*str*)

tbl2str(*tbl*, *sep_*)

str2tbl(*str*, *match_*, *ini_*)

lst2tbl(*lst*, *tbl_*)

tbl2lst(*tbl*, *lst_*)

8 Generic Functions

same(*a*, ...)

copy(*a*, ...)

tostring(*a*, ...)

totable(*a*, ...)

toboolean(*a*)

9 Special Functions

pause(*msg_*, *val_*)

atexit(*fun*, *uniq_*)

runonce(*fun*, ...)

collectlocal(*fun_*, *env_*)

Chapter 40. Generic Physics

Just a link (never written)

Chapter 41. External modules

Part V

PROGRAMMING

Chapter 42. MAD environment

Chapter 43. Tests

1 Adding Tests

Chapter 44. Elements

1 Adding Elements

Chapter 45. Commands

1 Adding Commands

Chapter 46. Modules

1 Adding Modules

2 Embedding Modules

Chapter 47. Using C FFI

Part VI

Indices and tables

- genindex
- modindex
- search

Bibliography

[ISOC99] ISO/IEC 9899:1999 Programming Languages - C. https://www.iso.org/standard/29237.html.

[XORSHFT03] G. Marsaglia, *"Xorshift RNGs"*, Journal of Statistical Software, 8 (14), July 2003. doi:10.18637/jss.v008.i14.

[TAUSWTH96] P. L'Ecuyer, *"Maximally Equidistributed Combined Tausworthe Generators"*, Mathematics of Computation, 65 (213), 1996, p203–213.

[MERTWIS98] M. Matsumoto and T. Nishimura, *"Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator"*. ACM Trans. on Modeling and Comp. Simulation, 8 (1), Jan. 1998, p3–30.

[CPXDIV] R. L. Smith, *"Algorithm 116: Complex division"*, Commun. ACM, 5(8):435, 1962.

[CPXDIV2] M. Baudin and R. L. Smith, *"A robust complex division in Scilab"*, October 2012. http://arxiv.org/abs/1210.4539.

[FADDEEVA] A. Oeftiger, R. De Maria, L. Deniau et al, *"Review of CPU and GPU Faddeeva Implementations"*, IPAC2016. https://cds.cern.ch/record/2207430/files/wepoy044.pdf.

[ISOC99CPX] ISO/IEC 9899:1999 Programming Languages - C. https://www.iso.org/standard/29237.html.

[MICADO] B. Autin, and Y. Marti, *"Closed Orbit Correction of Alternating Gradient Machines using a Small Number of Magnets"*, CERN ISR-MA/73-17, Mar. 1973.

[MATFUN] N.J. Higham, and X. Liu, *"A Multiprecision Derivative-Free Schur–Parlett Algorithm for Computing Matrix Functions"*, SIAM J. Matrix Anal. Appl., Vol. 42, No. 3, pp. 1401-1422, 2021.

Index

M

N

O

P

R

S

T

U

V

W

X