MAD-X expressions in Python (some ideas and prototypes)

R. De Maria

Thanks to F. Carlier, L. Deniau, G. Iadarola

MAD-X expressions in Python

MAD-X expressions are used to set dependencies between the variables and element parameters in sequence and optics files such as:

- circuit definitions
- layout constraints
- clever knobs to perform complicate operations

Not everything can be done in MAD-X, therefore t is useful then to extract this logic in other context, such as Python.

Problems:

1) How to implement data dependencies in Python
2) How to transfer the information from MAD-X to Python

There are several options with different objectives and constraints...

Data dependencies in Python

Many software deals with data dependencies using a variety of solutions and constraints.

To name a few Python packages:

- Tensorflow, pytorch, jax in the context of machine learning
- Dask, Luigi, joblib in the context of job managements
- TraitsUI in the context of user interfaces
- Many many other packages and contexts….

The goals in a context of flexible Python environment:

1) Implement data dependencies for any data structure, without needing to alter the data structure or source code: able to use it with any external code as well as internals
2) Sufficiently generic to model MAD-X style dependencies and LSA-style dependencies
3) Not slow down simulation codes

Simple option but...

```
!MAD-X code
c:=0.1*a+0.3*b;
d:=0.2*a+0.4*b;
```

Can work in simple cases but:
1) issues with names containing dots[1]
2) values are recomputed all the time
3) all values needs to go into special containers

Then
1) can be solved
2) impact on performance may be mitigated
3) cannot be avoided and requires any existing code to be rewritten

```python
class M:
    def __init__(self):
        self.values={}
        self.expr={}
    def __getitem__(self,k):
        ret=self.values.get(k,self.expr.get(k,0))
        if type(ret) is str:
            ret=eval(ret,{},self)
        return ret
    def __setitem__(self,k,v):
        if type(v) is str:
            self.expr[k]=v
        else:
            self.values[k]=v

if __name__=="__main__":
    m=M()
    m['c']="0.1*a+0.3*b"
    m['d']="0.2*a+0.4*b"

    print(m['c'])
    m['a']=3
    print(m['c'])
```

[1] name mangling is possible but cumbersome, rather using special parser...

Data dependencies as functions

Assuming values are stored in containers, a dependency can be viewed as task defined by functions using some input and modifying some output:

```
!MAD-X code
c:=0.1*a+0.3*b;
d:=0.2*a+0.4*b;
```

```
#MAD-X like behaviour
def seta(m,a):
    m.a=a
    m.c=0.1*m.a+0.3*m.b
    m.d=0.2*m.a+0.4*m.b

def setb(m,b):
    m.b=b
    m.c=0.1*m.a+0.3*m.b
    m.d=0.2*m.a+0.4*m.b
```

Data dependencies as functions

As dependencies becomes deeper this approach shows some complexity

```
!MAD-X code
c:=0.1*a+0.3*b;
d:=0.2*a+0.4*b;
e:=c+d;
```

```
#MAD-X like behaviour
def seta(m,a):
    m.a=a
    m.c=0.1*m.a+0.3*m.b
    m.d=0.2*m.a+0.4*m.b
    m.e=m.c+m.d

def setb(m,b):
    m.b=b
    m.c=0.1*m.a+0.3*m.b
    m.d=0.2*m.a+0.4*m.b
    m.e=m.c+m.d
```

```
def setc(m,c):
    m.c=c
    m.e=m.c+m.d
    #mod seta and setb

def setd(m,d):
    m.d=d
    m.e=m.c+m.d
    #mod seta and setb
```

Data dependencies as functions

Assuming values are stored in containers, a dependency can be viewed as task defined by functions using some input and modifying some output:

```
!MAD-X code
c:=0.1*a+0.3*b;
d:=0.2*a+0.4*b;
```

```
#MAD-X like behaviour
def seta(m,a):
    m.a=a
    m.c=0.1*m.a+0.3*m.b
    m.d=0.2*m.a+0.4*m.b

def setb(m,b):
    m.b=b
    m.c=0.1*m.a+0.3*m.b
    m.d=0.2*m.a+0.4*m.b
```

```
#LSA like behaviour
def seta(m,a):
    m.c+=0.1*(a-m.a)
    m.d+=0.2*(a-m.a)
    m.a=a

def setb(m,b):
    m.c+=0.3*(b-m.b)
    m.d+=0.4*(b-m.b)
    m.b=b
```

Data dependencies as tasks (1)

Target and dependencies are represented by references such as:

- `AttrRef(m,"a")` equiv to `m.a`
- `ItemRef(m,2)` equiv to `m[2]`
- Other types of refs are also possible such as file or row in database

A task has the methods:

- `get_targets()`, `get_dependencies()` returning set of references
- `run()` returning None (or e.g. a job handler for asynchronous jobs

A manager:

- contains a list of tasks
- when a set of targets is modified it runs the tasks needed to respect dependencies

Data dependencies as tasks (2)

In addition, with some operator overloading, references can also

- be factories that can generate references or expressions of references:
  - `mref=manager.ref(m);`
  - `mref.a` equiv to `AttrRef(m,"a",manager)`
  - `mref["a"]` equiv to `ItemRef(m,"a",manager);`
- be used to trigger the manager on set `ref.c=3`
- define tasks from expression:
  - `mref.c=0.1*mref.a+0.2*mref.a` or
  - `mref._expr("c=0.1*a+0.2*b")` or
  - `mref._madexpr("c:=0.1*a.b1+0.2*a.b2;")` for names with dots

A manager:

- could write setter functions to update quantities repeatedly without recomputing the correct ordering (if dependencies graph does not depend on values...)

Parsing MAD-X expression

Only MAD-X can parse MAD-X scripts: full language is ill defined, bugs and special cases…

...but `cpymad` can already gives parsed data structures! Only expressions needs to be parsed (unless re-build the MAD-X rpn evaluator…)

A grammar and parser generator (such as `lark`) are the most straightforward tool.

Here there is a solution to illustrate the idea.

Values, expression and elements from a full MAD-X script can be taken by inspecting `Madx` object in `cpymad`

```
from lark import Lark, Transformer, v_args

calc_grammar = """
    ?start: sum
        | NAME "=" sum      -> assign_var

    ?sum: product
        | sum "+" product   -> add
        | sum "-" product   -> sub

    ?product: power
        | product "*" atom  -> mul
        | product "/" atom  -> div

    ?power: atom
        | power "^" atom    -> pow

    ?atom: NUMBER           -> number
         | "-" atom         -> neg
         | "+" atom         -> pos
         | NAME             -> var
         | NAME "->" NAME   -> get
         | NAME "(" sum ("," sum)* ")" -> call
         | "(" sum ")"

    NAME: /[a-z_\.][a-z0-9_\.%]*/
    %import common.NUMBER
    %import common.WS_INLINE
    %ignore WS_INLINE
"""
```

```
@v_args(inline=True)
class MadxEval(Transformer):
    from operator import add, sub, mul, truediv as div
    from operator import neg, pos, pow
    number = float

    def __init__(self,variables,functions,elements):
        self.variables = variables
        self.functions = functions
        self.elements  = elements
        self.eval=Lark(calc_grammar, parser='lalr',
                       transformer=self).parse

    def assign_var(self, name, value):
        self.variables[name] = value
        return value

    def call(self,name,*args):
        ff=getattr(self.functions,name)
        return ff(*args)

    def get(self,name,key):
        return self.elements[name][key]

    def var(self, name):
        try:
            return self.variables[name.value]
        except KeyError:
            raise Exception("Variable not found: %s" % name)

def test():
    import math
    from collections import defaultdict
    madx=MadxEval(defaultdict(lambda :0),{},math)
    print(madx.eval("+1+2^-2"))
    print(madx.eval("a.b"))
    print(madx.eval("1+a.b*-3"))
    print(madx.eval("sin(3)^2"))
```

Parsing MAD-X data

```python
from cpymad.madx import Madx
mad=Madx()
mad.call("lhc.seq")
mad.call("optics.madx")

variables={}
for name,par in mad.globals.cmdpar.items():
    variables[name]=par.value

elements={}
for name,elem in mad.elements.items():
    elemdata={}
    for parname, par in elem.cmdpar.items():
        elemdata[parname]=par.value
    elements[name]=elemdata


import xdeps
import math

manager=xdeps.DepManager()
vref=manager.ref(variables)
eref=manager.ref(elements)
fref=manager.ref(math)
madeval=xdeps.MadxEval(vref,fref,eref).eval

for name,par in mad.globals.cmdpar.items():
    if par.expr is not None:
        vref[name]=madeval(par.expr)

for name,elem in mad.elements.items():
    for parname, par in elem.cmdpar.items():
        if par.expr is not None:
            if par.dtype==12: # handle lists
                for ii,ee in enumerate(par.expr):
                    if ee is not None:
                        eref[name][parname][ii]=madeval(ee)
            else:
                eref[name][parname]=madeval(par.expr)
```

1.5 sec

9.6 sec

17.1 sec

The LHC MAD-X environments contains:

- 2115 variables
- 13527 elements
- 62224 expressions!!!

Most of the expressions are some default expressions built by MAD-X for each element, not sure why...

```python
print(elements['mcbcv.5r1.b2']['kick'])
vref['on_x1']=2
print(elements['mcbcv.5r1.b2']['kick'])
```

```
1.6396535355504e-05
2.0495669193799964e-07
```

It updates 122 value

0.003 sec

xdeps

The code is in github.com/xsuites/xdeps, still at a prototype...

```
In [4]: manager.find_deps([vref['on_x1']])
Out[4]:
[_['on_x1'],
 _['acbxh2.l1'],
 _['mcbxh.2l1']['kick'],
 _['acbxh3.l1'],
 _['mcbxh.3l1']['kick'],
 _['acbch5.l1b2'],
 _['mcbch.5l1.b2']['kick'],
 _['acbyhs4.l1b1'],
 _['mcbyh.a4l1.b1']['kick'],
 _['acbyhs4.l1b2'],
 _['mcbyh.4l1.b2']['kick'],
 _['acbyvs4.l1b2'],
 _['mcbyv.a4l1.b2']['kick'],
 _['acbyhs4.r1b2'],
 _['mcbyh.a4r1.b2']['kick'],
 _['acbcv5.l1b1'],
 _['mcbcv.5l1.b1']['kick'],
 _['acbxv1.l1'],
 _['mcbxv.1l1']['kick'],
```

```
In [5]: manager.find_tasks([vref['on_x1']])
Out[5]:
[<_['acbxh2.l1'] = (((5.49019607843e-08*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((8e-06*_
 <_['mcbxh.2l1']['kick'] = _['acbxh2.l1']>,
 <_['acbxh3.l1'] = (((5.49019607843e-08*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((8e-06*_
 <_['mcbxh.3l1']['kick'] = _['acbxh3.l1']>,
 <_['acbch5.l1b2'] = ((((8.97888250373e-08*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((9.06
 <_['mcbch.5l1.b2']['kick'] = (-_['acbch5.l1b2'])>,
 <_['acbyhs4.l1b1'] = ((((-1.10276141294e-07*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((-2
 <_['mcbyh.a4l1.b1']['kick'] = _['acbyhs4.l1b1']>,
 <_['acbyhs4.l1b2'] = ((((2.42711053793e-07*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((-2.
 <_['mcbyh.4l1.b2']['kick'] = (-_['acbyhs4.l1b2'])>,
 <_['acbyvs4.l1b2'] = ((((9.24432833657e-08*_.sin(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((-3.
 <_['mcbyv.a4l1.b2']['kick'] = (-_['acbyvs4.l1b2'])>,
 <_['acbyhs4.r1b2'] = ((((-8.0129173092e-08*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((4.2
 <_['mcbyh.a4r1.b2']['kick'] = (-_['acbyhs4.r1b2'])>,
 <_['acbcv5.l1b1'] = ((((-8.97503400093e-08*_.sin(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((9.0
 <_['mcbcv.5l1.b1']['kick'] = _['acbcv5.l1b1']>,
 <_['acbxv1.l1'] = (((5.49019607843e-08*_.sin(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((-8e-06*
 <_['mcbxv.1l1']['kick'] = _['acbxv1.l1']>,
 <_['acbch5.r1b1'] = ((((1.0191848292e-07*_.cos(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((-1.02
 <_['mcbch.5r1.b1']['kick'] = _['acbch5.r1b1']>,
 <_['acbxv2.l1'] = (((5.49019607843e-08*_.sin(((_['phi_ir1']*_['twopi'])/360.0)))*_['on_x1'])+((-8e-06*
 <_['mcbxv.2l1']['kick'] = _['acbxv2.l1']>,
```

Loose points: testing, extend to dynamic topologies, loops, code generation, speed-ups, syntax cleanup

Other example, ideas

```
import xdeps

manager=xdeps.DepManager()
m=manager.refattr(globals(),'m')

a=b=0
m.c=0.1*m.a+0.2*m.b
m.a=2
print(c)
m.b=3
print(c)
```

```
0.2
0.8
```

Alternative ref that return always Item ref and give a default values if missing