
CERN

Pythonic masks and selected examples

G. Sterbini, G. Iadarola

Thanks to R. De Maria, S. Fartoukh, F. Van Der Veken, S. Kostoglou, A. Poyet

Introduction

- As complement of previous presentation, we will focus on some simplified examples to present and discuss the potential of the pythonic approach for the MAD-X masks.
- The complete notebook and the needed files are available here (there you can find many more examples).

The simplest piece of code

```
from cpymad.madx import Madx
mad = Madx(command_log='my_log.log')
mad.input('''
option, echo=false, warn=false;
call, file=before_luminosity_leveling.save;
option, echo=true, warn=true;
use, sequence=lhcb1;
''')
mad.twiss()
```

1. Full back-compatibility in the logging file

```
option, echo=false, warn=false;
call, file=before_luminosity_leveling.save;
option, echo=true, warn=true;
use, sequence=lhcb1;

twiss;
```

2. 1-to-1 translation from MADX using input()

3. Useful wrapper functions

Functions and dataframes

Possibility to add/customize/document new methods (vs MAD-X macros)

```
def set_survey_twiss(self, sequence):
    'To set a sequence, survey it and twiss it.'
    self.input(f'set, sequence={sequence};survey; twiss;')

Madx.set_survey_twiss=set_survey_twiss
mad.set_survey_twiss(sequence='lhcb1')
```

Availability of modern data structures (vs MAD-X tables)

```
mad.table['summ'].dframe()
```

OUTPUT

length	orbit5	alfa	gammatr	q1	dq1	betxmax	dxmax	dxrms	xcomax	...
26658.8832	-0.0	0.000348	53.598201	62.309979	1.512814	6742.132344	3.273475	1.40695	0.012156	...

- Migrate the scripting logic of the mask **in python and test it with the *python debugger*.**
- Increase usability/synergies **with other analysis tools (e.g., *pandas*).**

Variables inspection

MAD-X has a memory global space: essential feature for the knobs but a hassle for the user and for the debugging.

Monitor changes on the MAD-X global space

```
globals_before=deepcopy(dict(mad.globals))
mad.globals['cmrs.b1_sq']+=.001
globals_after=deepcopy(dict(mad.globals))
```

Knob controlling the coupling

OUTPUT

```
The following variables were modified:
['kqs.r3b1', 'cmrs.b1_sq', 'kqs.l4b1', 'kqs.a67b1', 'kqs.r7b1', 'kqs.l8b1', 'kqs.a23b1']
```

List constants, independent and dependent variables

```
my_global=mad.get_variables_dataframes()
constants=my_global['constants']
independent_variables=my_global['independent_variables']
dependent_variables=my_global['dependent_variables']
dependent_variables[['value','expression','knobs']].loc[['kqs.a23b1','kqs.a45b1']]
```

OUTPUT

	value	expression	knobs
kqs.a23b1	0	(0.000000000000e+00)*ona2_b1+(0.142516736842e-...	[cmis.b1, cmis.b1_sq, cmrs.b1, cmrs.b1_sq, ona...
kqs.a45b1	0	(0.000000000000e+00)*ona2_b1+(0.113812285983e-...	[cmis.b1, cmis.b1_sq, cmrs.b1, cmrs.b1_sq, ona...

Listing knob dependencies

Variables dependent on a given knob

```
def get_knob_dependence(knob_name, dataframe):
    return dataframe[dataframe.apply(lambda x: knob_name in x['knobs'], axis=1)]

get_knob_dependence(knob_name='cmrs.b1_sq', dataframe=dependent_variables).index
```

OUTPUT

```
['kqs.a23b1', 'kqs.a45b1', 'kqs.a67b1', 'kqs.a81b1', 'kqs.l2b1',
 'kqs.l4b1', 'kqs.l6b1', 'kqs.l8b1', 'kqs.r1b1', 'kqs.r3b1', 'kqs.r5b1',
 'kqs.r7b1'],
```

Sequence elements dependent on a given knob

```
lhcb1_df=mad.get_sequence_df('lhcb1')
get_knob_dependence(knob_name='cmrs.b1_sq',dataframe=lhcb1_df).index
```

OUTPUT

```
['mqs.23r3.b1', 'mqs.27r3.b1', 'mqs.27l4.b1', 'mqs.23l4.b1',
 'mqs.23r4.b1', 'mqs.27r4.b1', 'mqs.27l5.b1', 'mqs.23l5.b1',
 'mqs.23r5.b1', 'mqs.27r5.b1', 'mqs.27l6.b1', 'mqs.23l6.b1',
 'mqs.23r6.b1', 'mqs.27r6.b1', 'mqs.27l7.b1', 'mqs.23l7.b1',
 'mqs.23r7.b1', 'mqs.27r7.b1', 'mqs.27l8.b1', 'mqs.23l8.b1',
 'mqs.23r8.b1', 'mqs.27r8.b1', 'mqs.27l1.b1', 'mqs.23l1.b1',
 'mqs.23r1.b1', 'mqs.27r1.b1', 'mqs.27l2.b1', 'mqs.23l2.b1',
 'mqs.23r2.b1', 'mqs.27r2.b1', 'mqs.27l3.b1', 'mqs.23l3.b1']
```

Knob-independent functions vs macros

```
mad = Madx(command_log="my_log.log")
mad.input('''option, echo=false, warn=false;
call, file=before_luminosity_leveling.save;
use, sequence=lhcb1;''')
coupling_measurement(mad, qx=62.313, qy=60.318,
                     tune_knob1_name='dqx.b1_sq',
                     tune_knob2_name='dqy.b1_sq',
                     sequence_name='lhcb1',skip_use=False)
```

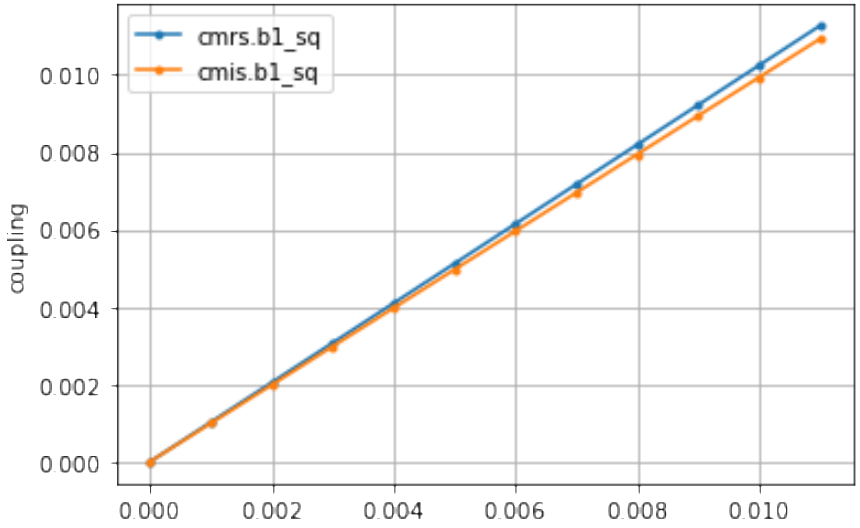
OUTPUT

```
1.827918083563418e-07
```

→ Improving **modularity**
→ **Maintaining compatibility**: single implementation for different knob naming conventions (e.g., Run3 and HL)

Multiple MAD-X instances

```
import multiprocessing
import numpy as np
pool = multiprocessing.Pool(4)
tune_knob1_name='dqx.b1_sq'
tune_knob2_name='dqy.b1_sq'
# coupling_knob='cmrs.b1_sq' or 'cmis.b1_sq'
def to_compute_coupling(coupling_knob_value, coupling_knob):
    mad = Madx()
    mad.input('option, echo=false, warn=false;')
    mad.input('call, file="before_luminosity_leveling.save";')
    mad.input('use, sequence=lhcb1;')
    ref=mad.globals[coupling_knob]
    print(f'STEP {coupling_knob}={coupling_knob_value}')
    mad.globals[coupling_knob]=ref+coupling_knob_value
    my_coupling=coupling_measurement(mad, qx=62.313, qy=60.318,
                    tune_knob1_name=tune_knob1_name,
                    tune_knob2_name=tune_knob2_name,
                    sequence_name='lhcb1',skip_use=True)
    return my_coupling
```



Multiple instances in the same script
→ Parallelization (as shown)
→ Key-ingredient for beam-beam for B4

Luminosity computation

→ Computing luminosity is an example of a scripting task that could be moved from MAD-X to python.

1. Get bunches and optics parameters from MAD-X

```
for my_sequence in ['lhcb1', 'lhcb2']:
    mad.use(my_sequence)
    mad.twiss()
    twiss_dict[my_sequence]=mad.get_twiss_df('twiss')
    bunch_dict[my_sequence]=dict(mad.sequence[my_sequence].beam)
```

2. Customize/overwrite parameters

```
bunch_frequency=bunch_dict['lhcb1']['freq0']
bunch_dict['lhcb1']['npart']=1.1e11
bunch_dict['lhcb2']['npart']=1.1e11
```

3. Compute luminosity with the "luminosity library"

```
result=compute_luminosity('ip8:1', my_filling_pattern.n_coll_LHCb, bunch_frequency,
                          bunch_dict['lhcb1'],bunch_dict['lhcb2'],
                          twiss_dict['lhcb1'], twiss_dict['lhcb2'])
print(f'Luminosity in IP8: {result} [Hz/cm^2]')
```

OUTPUT

```
Luminosity in IP8: 3.434686535862147e+31 [Hz/cm^2]
```

Luminosity leveling

→ Once we can compute luminosity, we can invert it (via *least-square*) to lumi-level.

1. Chose a luminosity target

```
L_target=2e+33
```

2. Define the function to minimize. Here you define where to level (e.g., IP8) and the variable to be used for leveling (e.g., vertical separation).

```
def function_to_minimize(delta):
    aux=get_luminosity_dict('ip8:1', my_filling_pattern.n_coll_LHCb,
                            bunch_frequency, bunch_dict['lhcb1'], bunch_dict['lhcb2'],
                        twiss_dict['lhcb1'], twiss_dict['lhcb2'])
    aux['y_1']=-delta
    aux['y_2']=delta
    return pm.luminosity(**aux)-L_target
```

3. Minimization that is luminosity leveling.

```
aux=least_squares(function_to_minimize, starting_guess, bounds=(0, 3e-4))
print(aux)
print(f"\nLuminosity after ideal levelling: {function_to_minimize(aux['x'][0])+L_target} Hz/cm^2")
```

4. Set the knob using the corresponding metric.

```
mad.globals['on_sep8v']=-aux['x'][0]*1000
```

5. Sanity check that the inversion was correctly done.

```
result=compute_luminosity('ip8:1', my_filling_pattern.n_coll_LHCb, bunch_frequency,
                          bunch_dict['lhcb1'],bunch_dict['lhcb2'],
                          twiss_dict['lhcb1'], twiss_dict['lhcb2'])
print(f'Luminosity in IP8: {result} [Hz/cm^2]')

# Verify with assertion
assert L_target*0.99<result<L_target*1.01
```

OUTPUT

```
Luminosity in IP8: 2.000540358695341e+33 [Hz/cm^2]
```

BB bunch-by-bunch analysis

→ In pymask, as discussed, we use dataframes to define/customize/install BB lenses.

```
mad = pm.Madxp(command_log="my_log.log")
mad.input('option, echo=false, warn=false;')
mad.input('call, file="before_luminosity_leveling.save";')
mad.input('option, echo=true, warn=true;')
mad.use('lhcb1')
bb_df=pd.read_pickle('bb_df_track.pickle')
pm.install_lenses_in_sequence(mad, bb_df, 'lhcb1')
mad.use('lhcb1')
mad.globals.on_bb_charge = 0
```

One can use python scripting capability to edit the BB dataframe, e.g., for bbb computation.

BB bunch-by-bunch analysis (I)

STEP 1: download the data from (NX)CALS

```
variables=['LHC.BCTFR.A6R4.B%:BUNCH_FILL_PATTERN', 'LHC.BCTFR.A6R4.B%:BUNCH_INTENSITY']
startTime = pd.Timestamp('2018-10-09 19:30', tz='CET')
endTime = pd.Timestamp('2018-10-09 19:31', tz='CET')
raw_data = importData.cals2pd(variables,startTime,endTime)
```

STEP 2: build-up the filling pattern

```
my_filling_pattern= fp.FillingPattern(raw_data['LHC.BCTFR.A6R4.B1:BUNCH_FILL_PATTERN'].dropna().iloc[0],
                      raw_data['LHC.BCTFR.A6R4.B2:BUNCH_FILL_PATTERN'].dropna().iloc[0])
```

STEP 3: choose the bunch to track (B1 in this example)

```
bunch_to_track=18
```

STEP 4: edit the BB dataframe to select only encounters relevant to the bunch to track

```
bb_df_filtered=filter_bb_df(bb_df, my_filling_pattern.b1.bb_schedule.loc[bunch_to_track])
```

STEP 5: scale the BB intensities accordingly to the B2 partner bunch measured intensities

```
B2_bbb_intensity=raw_data['LHC.BCTFR.A6R4.B2:BUNCH_INTENSITY'].dropna().iloc[0]
bb_df_scaled=bbb_rescaling_bb_df(bb_df_filtered, my_filling_pattern.b1.bb_schedule.loc[bunch_to_track],
                                 B2_bbb_intensity, reference_intensity=1.8e11)
```

Bunch-by-bunch analysis

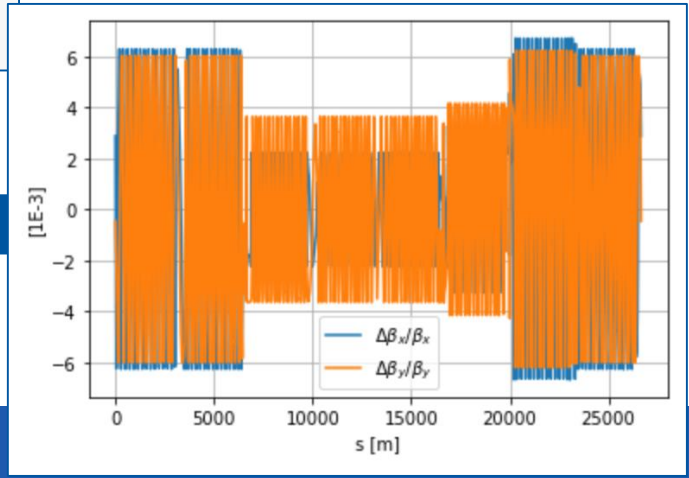
STEP 6: re-generate BB info.

```
pm.generate_mad_bb_info(bb_df_scaled, mode='from dataframe')
```

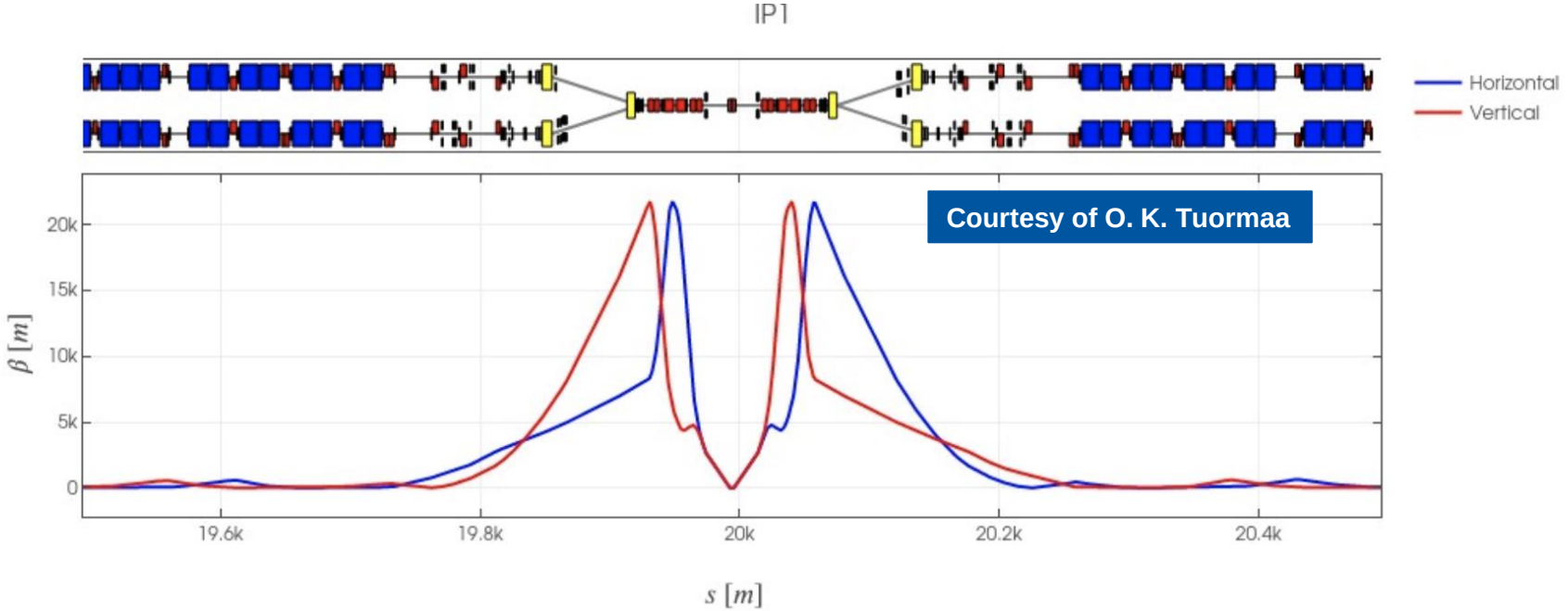
STEP 7: install the BB from the edited BB dataframe.

```
mad = pm.Madxp(command_log="my_log.log")
mad.input('option, echo=false, warn=false;')
mad.input('call, file="before_luminosity_leveling.save";')
mad.input('option, echo=true, warn=true;')
mad.use('lhcb1')
pm.install_lenses_in_sequence(mad, bb_df_scaled, 'lhcb1')
```

STEP 8: e.g., plot beta-beating of bunch # 18 of B1.



On plotting capability…and synergies…



Other colleagues (TE-MPE) are using the pymask approach for fast beam failure studies
→ https://gitlab.cern.ch/machine-protection/fast-beam-failures

Summary

We showed via simple examples how to use **MAD-X as a optics library** and **move the mask scripting layer to python**.

What could we gain?

- **Inspections** of memory global space and knob dependences
- Improved **modularity and debug/document** capability
- Gluing **multiple MAD-X instances** (essential for our BB B4 approach)
- Improved **compatibility** across the optics
- Systematic/automatic **sanity checks**
- Profiting of **momentum outside** CERN (e.g., cpymad, pandas, …)
- Profiting of **momentum inside** CERN (e.g., pytimber, luminosity model…)

Thank you for your attention.


CERN