# PyMad – INTEGRATION OF MadX IN PYTHON

K. Fuchsberger, Y. Inntjore Levinsen (CERN, Geneva, Switzerland)

*Abstract*

MadX (Methodical Accelerator Design), a very commonly used software package to model particle accelerators, is implemented and still maintained in the programming languages C and FORTRAN. For detailed processing, analysis and plotting of MadX results, other programming languages are often used. One very popular scripting language is Python, which is widely used in the physics community and provides powerful numerical libraries and plotting routines. Therefore, access to MadX models from Python is a common demand. Currently, several possible concepts for the realisation of such a project are evaluated, including direct access to MadX via Cython (C extensions for Python) or the re-usage of the existing JMad Java libraries, benefiting from the already available model-definitions. A first prototype is already in use and we have released the code under an open source license. This paper presents the concepts and the current status of the project, as well as some usage examples.

## MOTIVATION

MadX [1] is the latest iteration of the highly successful MAD program code, used by a very large community at CERN and elsewhere. Numerous lattice models exist for most accelerators at CERN, including the SPS, the LHC and the transfer lines which are regularly maintained and updated. MadX is designed as a standalone software with its own scripting language, which is used to interface with the software.

Although this MadX-language contains many elements of a scripting language (like loops or if/else statements) it is not (and was never intended to be) a full programming language with custom libraries. Therefore the necessity arises to post-process output data with other tools, especially when doing complex simulation tasks. The typical way of using MadX from a higher level programming language other then MadX scripting is:

1. Create an input file for MadX (ASCII file) containing model definition calls, input parameters and commands to export the results.
2. Call MadX with the created input file.
3. Wait until MadX terminates.
4. Parse the MadX output files.
5. Postprocess the data (e.g. plot).

Although this can be easily done because of the highly configurable MadX text file output features, it has several disadvantages, including:

- Creating MadX files by simply composing strings as demanded by the application is very error-prone and makes the application code very dependent on the MadX scripting language as well as on the model.
- When MadX is ran with input file, it terminates when it has finished. Since this also requires to load the sequence (model definition) it becomes a very time consuming procedure, especially when many iterations are needed (e.g. for fitting purposes).
- Every application developer ends up in implementing its own MadX parser.

All these disadvantages can be avoided if steps 1 to 4 are encapsulated in a dedicated software package with a well defined API in a higher level programming language. Starting and stopping of MadX can be avoided by keeping a running instance with the actual model status in memory. All the communication can then be done in the language-typical way which allows proper error handling and the usage of language specific libraries.

This concept has already proven useful in the form of JMad[1] [2, 3], an API for the Java programming language which is used by different simulation tools and LHC online applications. The need for a similar API for the Python programming language arises from the fact that this language is widely used in the physics community and that sophisticated numerical and plotting libraries exist for Python.

## DESIGN GOALS

Since every programming language has its own characteristics, an API for one programming language does not necessarily fit directly in another language. For JMad a lot of effort was put in type safety to detect errors already at compile time. Since Python is a scripting language, this is not possible. On the contrary, Python users expect a very flexible set of commands and parameters, and checks for type safety at run time. In that sense Python fits nicely the approach of the MadX proprietary scripting language, where most of the function parameters are optional. While both Python and Java are object oriented languages, it is common in Python to provide convenience functions that (at least partially) hide the underlying object oriented structure. When designing PyMad, we tried to follow similar principles.

Another important aspect related to the usage of MadX is what we call 'model definitions'. A model definition provides the MadX script files needed to setup a valid model for a certain accelerator together with the knowledge in which order to call them and what options are available (e.g. different optics or sequences). During the development of JMad, a lot of effort was made to establish a simple work-flow to create and maintain such model definitions.

[1] http://cern.ch/jmad

Therefore, it is highly desirable to reuse these definitions in other libraries. This ensures that the same models are used for simulations with different libraries and that the results are comparable.

## OVERVIEW

To fulfil the goals mentioned in the previous section, two ideas were born:

- Compile MadX as a shared library and use Cython to access the internal functions.
- Access a running instance of JMad and wrap some Python-like functionality around it.

Both concepts have certain advantages that we will outline later. To explore the possibilities of either one and at the same time establish a user friendly API, we decided to create a common project. The components involved in this project are outlined in Fig. 1.

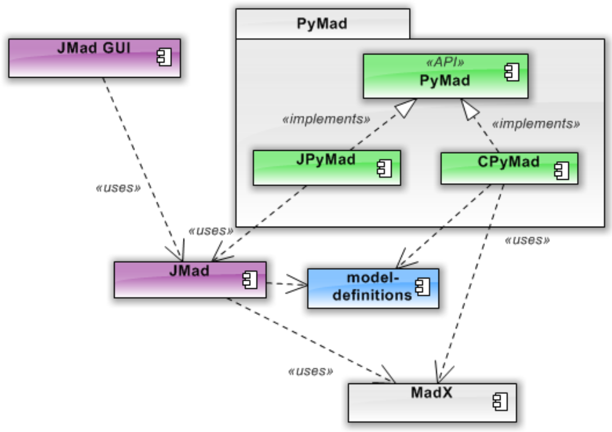

Figure 1: Overview of PyMad components.

The main idea of PyMad is to provide one API which can be used in two different modes. As visible in Fig. 1, the main API is named PyMad. The implementation which uses the existing JMad infrastructure is denoted JPyMad while the one directly accessing MadX as a shared library is named CPyMad. If user scripts only use the common API, then they can be run against both of the implementations, choosing by a simple switch which one to use.

## USAGE

### *Installation*

Depending on the intended usage (CPyMad, JPyMad or JPyMad with GUI), different installation steps are necessary. Detailed instructions can be found on the PyMad website[2]. The sourcecode of PyMad was released under the Apache 2.0 License[3] and is currently available on Github[4].

[2]http://www.cern.ch/pymad
[3]http://apache.org/licenses/
[4]https://github.com/pymad/pymad

### *Example scripts*

Listing 1 shows a typical example: First the PyMad environment is initialised. This is done by calling the function `init()`. The first parameter of this function is a string which defines the mode to use (`'cpymad'` or `'jpymad'`). All the rest of the parameters which can be given here depend on the mode (For example in the listing the paramter `start='gui'` is used, which instructs JPyMad to start the java graphical user interface). The init function returns a `PyMadService` object. This service provides methods to list the available model definitions and create and delete models. Depending on the implementation, a `PyMadService` can have one or more running models. In the example then a new model (with the model-definition name 'lhc') is created. Then the twiss functions are calculated for this model and plotted using Matplotlib.

**Listing 1: plot_beta.py**

```
from matplotlib import pyplot as plt
import pymad as pm

# create the service
#pms = pm.init('cpymad')
pms = pm.init('jpymad', start='gui')

# create a model by name of model-definition
model = pms.create_model('lhc')

# obtain get twiss table in a python object
table, summary = model.twiss(columns=['name',
     's', 'betx', 'bety'])

# plot the result
plt.plot(table.s, table.betx)
plt.show()

# cleanup everything (e.g. close gui)
pms.cleanup()
```

Although it is good practice to use the service returned by the init method, this is not absolutely necessary in all cases: Internally, the service is kept as a singleton instance and the PyMad API provides several convenience methods which then access this singleton per default. This is demonstrated in listing 2, which prints out all the available model definitions provided by the service and all the currently running models.

**Listing 2: ls_info.py**

```
import pymad as pm

#pm.init('cpymad')
pm.init('jpymad', start='gui')

# print the name of all model definitions
pm.ls_mdefs()

# list the available (running) models
pm.ls_models()

pm.cleanup()
```

## IMPLEMENTATION DETAILS

### *JPyMad*

This implementation arose from experiences during the development of JMad. The key idea was to use as much as possible from the JMad infrastructure and only implement the top components which define the API for Python scripts. We chose to use Py4j[5] as bridge between java and Python. Py4j allows Python to access objects from a running java instance and also java programs to call Python scripts. It does so by creating a TCP connection between an object on the java side (called a `GatewayServer`) and one on the Python side (called a `JavaGateway`). The advantage of using this technology is that all the existing Python libraries can be used. This would not be possible if one would e.g. use Jython, which is a Python implementation which runs in the java virtual machine itself.

### *CPyMad*

Python has an efficient syntax and garbage collection, but as a scripting language it suffers from slowness in many cases. Using compiled libraries in the background with Python as the front end can drastically improve the performance. A framework that simplifies this approach is Cython. Cython is a Python implementation with added C definitions. A dynamically compiled binary is created which can then be imported into a normal Python code. Cython is an excellent tool to integrate existing C libraries in higher level Python routines. It can also be used to speed-up existing Python code by declaring e.g. loop variable as C type variables instead of the slower Python declared variables.

**Listing 3: madx_structures.pxd: sequence & sequence_list**

```
cdef extern from "madx.h":
  struct sequence:
    char[48] name
    pass    # ignore the rest of the struct
  struct sequence_list:
    sequence **sequs
    int curr
    pass
```

We use these features for the second implementation, named CPyMad. In CPyMad we use a MadX as a shared library together with Cython files which define the accessible structures and functions in MadX. This implementation provides the very interesting feature that we can directly access the memory of MadX, not having to go through files. This can give significant gains on slow network disks or if you want to load large tables of data.

**Listing 4: Print Sequence Names**

```
from cpymad.madx_structures cimport
    sequence_list
# Mad-X header file..
cdef extern from "madextern.h":
 sequence_list *madextern_get_sequence_list()

# this can be called in a Python script.
def print_sequence_names():
 cdef sequence_list *seqs
 seqs= madextern_get_sequence_list()
 for i in xrange(seqs.curr):
  print("seqs.sequs[i].name")
```

[5] `http://py4j.sourceforge.net`

For illustrative purposes, we show in Listing 3 a small part of the structures defined in MadX. Listing 4 shows the minimal code needed to generate a function that prints the sequence names currently available in MadX, based on these structures.

### *Comparison*

Both implementations have their own advantages and disadvantages and specialities: Since CPyMad accesses the MadX model directly, it is expected to perform better than the JPyMad implementations, whose communication with MadX is based on pipes and files. Both JPyMad and CPyMad can run an arbitrary amount of models in parallel, but it requires CPyMad to use multiprocessing (currently implemented for the models). Advantages for JPyMad are the existing Java GUI, which can be used in parallel to the Python scripts, and the many already existing model definitions. We hope in the near future to converge on the model definitions in JMad and CPyMad.

Currently, CPyMad can only be installed on Linux and OSX operating systems, since a dynamically linked library of MadX is not available for Windows. JPyMad runs on all three platforms. While JPyMad requires a Java runtime installed on the system, CPyMad does not.

## SUMMARY AND OUTLOOK

Already in this early stage of implementation, PyMad is a useful tool that comfortably combines the power of MadX simulations and Python libraries. PyMad provides a clean API with two different implementations: One (CPyMad), which accesses the memory of MadX directly and one (JPyMad) which uses the existing JMad infrastructure. The next steps for JPyMad will be to expose more and more JPyMad features (e.g. object oriented access to the model-elements) to the Python API. For CPyMad it will be important to provide a mechanism to also use the JMad model definitions to guarantee the comparability of results. Further, work on the readout of the twiss table from MadX memory is ongoing, which finally will replace the current data-transfer between Python and MadX and therefore is expected to drastically improve the speed.

## REFERENCES

[1] W. Herr, F. Schmidt “A MAD-X Primer”, CERN AB Note, CERN-AB-2004-027-ABP.

[2] K. Fuchsberger et al., “JMad - Integration of MadX into the Java World”, proceedings IPAC 2010.

[3] K. Fuchsberger et al., “Status of JMad, the Java-API for MadX”, these proceedings.