pymadx Documentation

Release 2.0.1.dev0+ga0f8113.d20230316

Royal Holloway

Mar 16, 2023

CONTENTS

pymadx is a set of classes and functions to load MADX output as well as prepare MADX models. The package overall functions as a holder for any code required to load and manipulate MADX output data as well as prepare MADX and PTC models programmatically.

CHAPTER

ONE

LICENCE & DISCLAIMER

1.1 CERN MADX

This software is not officially endorsed by the MADX team at CERN and is not related to any similarly named software provided by CERN. It has been developed purely as a utility for BDSIM.

1.2 Licence

CHAPTER

TWO

AUTHORSHIP

The following people have contributed to pymadx:

- Laurie Nevay
- Andrey Abramov
- Stewart Boogert
- William Shields
- Jochem Snuverink
- Stuart Walker

CHAPTER

THREE

INSTALLATION

3.1 Requirements

- pymadx is developed exclusively for Python 3. 3.7 is the minimum version.
- Matplotlib
- Numpy

3.2 Installation

pymadx can be installed using pip with internet access without downloading the git repository:

::

pip install pymadx

Alternatively, ff cloning the git repository and installing locally, a set of useful commands are provided in a simple Makefile included in the main directory. In this case, to install pymadx, simply run `make install` from the root pymadx directory.:

```
cd /my/path/to/repositories/
git clone http://bitbucket.org/jairhul/pymadx
cd pymadx
make install
```

Alternatively, run `make develop` from the same directory to ensure that any local changes are picked up.

3.3 For Developers

If you want to create a package that depends on pymadx, it has the optional component pytransport that can be requested as `pymadx[pytransport]` in the *pyproject.toml*.

CHAPTER

FOUR

TFS FILE LOADING & MANIPULATION

MADX outputs Twiss information as well as PTC tracking data in their own Table File System (TFS). This is the only format used by MADX. pymadx includes a class called Tfs for the purpose of loading and manipulating this data.

The TFS format is described in the MADX manual available from the madx website. The format roughly is described as a text file. The file contains first a header with key and value pairs for one-off definitions. This is proceeded by a line with column names and a line with the data type of each column. After this each line typically represents the values of the lattice for a particular element with each new line containing the values at a subsequent element in the lattice. We maintain the concept of this table and refer to 'rows' and 'columns'.

4.1 Tfs Class Features

- Loading of TFS files.
- Loading of TFS files compressed and archived with .tar.gz suffix without decompressing.
- Report a count of all different element types.
- Get a particular column.
- Get a particular row.
- Get elements of a particular type.
- Get a numerical index from the name of the element.
- Find the curvilinear S coordinate of an element by name.
- Find the name of the nearest element at a given S coordinate.
- Plot an optics diagram.
- Roll a lattice to start from a different point.
- Calculate a beam size given the Twiss parameters, dispersion and emittance (in the header).
- Determining whether a given component perturbs the beam.
- Extract a 'segment' if PTC data is present.
- Slice a lattice (in the Python sense) with new S coordinates.

4.2 Loading

A file may be loading by constructing a Tfs instance from a file name.

```
>>> import pymadx
>>> a = pymadx.Data.Tfs("myTwissFile.tfs")
```

Note: The import will be assumed from now on in examples.

A file compressed using tar and gzip may also be loaded without first uncompressing without any difference in functionality. Not temporary files are created:

```
tar -czf myTwissFile.tar.gz myTwissFile.fs
```

```
>>> import pymadx
>>> a = pymadx.Data.Tfs("myTwissFile.tar.gz")
```

Note: The detection of a compressed file is based on 'tar' or 'gz' existing in the file name.

4.3 Twiss File Preparation

You may export twiss data from MADX with a choice of columns. We often find it beneficial to not specify any columns at all, which results in all available columns being written. This large number (~70) makes the file less human-readable but ensures no information is omitted. Such an export will also increase the file size, however, we recommend compressing the file with gzip as ASCII files compress very well with a typically compression ratio of over 10:1.

The following MADX syntax in a MADX input script will prepare a Tfs file with all columns where "SEQUENCENAME" is the name of the sequence in MADX.:

```
select,flag=twiss, clear;
twiss,sequence=SEQUENCENAME, file=outputfilename.tfs;
```

4.4 Querying

The Tfs class can be used to query the data in various ways.

4.4.1 Basic Information

- All data is stored in the **data** object inside the class
- The header information is stored in **header**.
- The names of all elements in order is stored in **sequence**.
- The names of all columns in the file is stored in **columns**

Generally, members beginning with small letters are objects and capital letters are functions.

A nice summary of the file can be provided with the *ReportPopulations* function.:

```
a = pymadx.Data.Tfs("mytwissfile.tar.gz")
a.ReportPopulations()

Filename > twiss_v5.2.tfs
Total number of items > 1032
Type........... Population
MULTIPOLE...... 516
DRIFT.......... 201
QUADRUPOLE..... 102
MARKER......... 78
MONITOR........ 64
SBEND.......... 24
SEXTUPOLE...... 18
HKICKER........ 15
VKICKER........ 14
```

4.4.2 Indexing and Slicing

The instance may be indexed like a normal Python iterable structure such as a list or a tuple. Square brackets with a number *i* will return the *ith* element in the sequence. A Python 'slice' may also be used where a range of elements may be selected. If only one element is indexed a Python dictionary is returned for that element. If a range is required, another Tfs instance is returned:

```
a = pymadx.Data.Tfs("mytwissfile.tar.gz")
a[3]          # 4th element in sequence (0,1,2,3!)
a[3:10]       # 4th to 11th elements (tfs instance returned)
a['IP1':300] # find element named IP1 (exactly) and start from that until the #301th␣
↪element
a['IP3':]     # find element named IP3 (exactly) and take from there to the end of the␣
↪file
a['L230A']    # returns a Python dictionary for element named L230A
```

If you know the name of an element you can search for it and get the index from that.:

```
a.IndexFromName('L230A')
>>> 995
```

You can also search by nearest curvilinear S coordinate along the beam line.:

```
a.IndexFromNearestS(34.4)
>>> 225
a[225]['NAME']
```

4.4.3 Row or Element

A row of data is an entry for a particular element. The Tfs class is conceptually a list of elements. Each element is represented by a Python dictionary that has a key for each column. The list of acceptable keys (ie names of columns) can be found in the member named 'colums'.:

```
a.columns #prints out list of column names
```

If a single element is indexed, a dictionary is returned and can be accessed - even in one step.:

```
d = a[123]
d['NAME']
>>> 'MQD8X'
a[123]['NAME'] # equivalent
```

4.4.4 Looping & Iterating

The Tfs class may be iterated over like a list in Python. For each iteration a dictionary for that element is returned.:

```
for el in a:
    print(el['NAME'])
```

4.4.5 Beam Sizes

For convenience the beam size is calculated from the Beta amplitude functions, the emittance and dispersion if they are present. The emittance is defined by 'EX' and 'EY' in the header. These are calculated according to

$$\sigma_x = \sqrt{\beta_x \epsilon_x + D(S)^2 \frac{\sigma_E^2}{\beta_{\mathrm{Lorentz}}^2}}$$

$$\sigma_y = \sqrt{\beta_y \epsilon_y + D(S)^2 \frac{\sigma_E^2}{\beta_{\mathrm{Lorentz}}^2}}$$

σ_E in MADX is fractional. Here we use the relation

$$\sigma_E = \frac{\Delta E}{E} = \beta_{\mathrm{Lorentz}}^2 \frac{\Delta p}{p}$$

Note: MADX input files often don't have a sensible emittance defined as it is not always required. Ensure the emittance is what you intended it to be in the Tfs file.

4.4.6 Modification

It is not recommended to modify the data structures inside the Tfs class. Of course one can, but one must be careful of Python's copying behaviour. Often a 'deep copy' is required or care must be taken to modify the original and not a reference to a particular variable.

CHAPTER

FIVE

PLOTTING

The *pymadx.Plot* module provides various plotting utilities.

5.1 Plotting Features

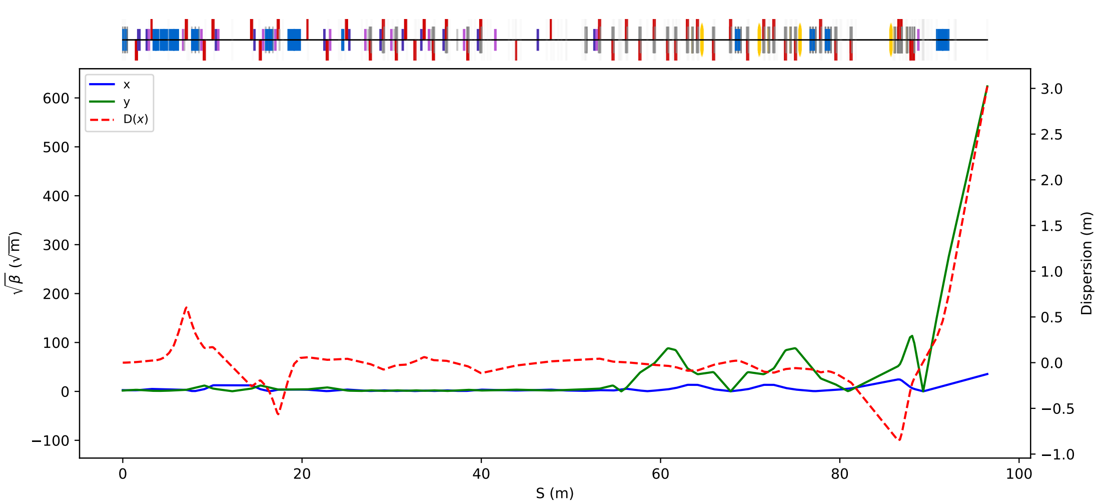
- Make default optics plots.
- Add a machine lattice to any pre-existing plot.
- Interactive plots with the machine diagram following the mouse zooming.
- Interactive plots with searching for nearest element.

5.2 Optics Plots

A simple optics plot may be made with the following syntax:

```
a = pymadx.Data.Tfs("mytwissfile.tar.gz")
a.Plot()
```

This creates a plot of the Beta amplitude functions against curvilinear S position. A colour diagram representing the machine is also produced above the graph as shown below.



The command has optional arguments such as a title string to be put at the top of the graph and whether to also plot the horizontal dispersion function. This function is provided as a quick utility and not the ultimate plotting script. The user can make their own plot and then append a machine diagram at the end if they wish.:

```
f = matplotlib.pyplot.figure()
# user plotting commands here
pymadx.Plot.AddMachineLatticeToFigure(f, "mytwissfile.tar.gz")
```

gcf() is a matplotlib.pyplot function to get a reference to the current matplotlib figure and can be used as the first argument.:

```
pymadx.Plot.AddMachineLatticeToFigure(gcf(), "mytwissfile.tar.gz")
```

Note: It becomes difficult to adjust the axes and layout of the graph after adding the machine description. It is therefore strongly recommended to do this last.

5.3 Colour Coding

Each magnet is colour coded an positioned depending on its type and strength.

Type	Shape	Colour	Vertical Position
drift	N/A	Not shown	N/A
sbend	Rectangle	Blue	Central always
rbend	Rectangle	Blue	Central always
hkicker	Rectangle	Purple	Central always
vkicker	Rectangle	Pink	Central always
quadrupole	Rectangle	Red	Top half for K1L > 0; Bottom half for K1L < 0
sextupole	Hexagon	Yellow	Central always
octupole	Hexagon	Green	Central always
multiple	Hexagon	Light grey	Central always
rcollimator	Rectangle	Black	Central always
ecollimator	Rectangle	Black	Central always
any other	Rectangle / Line	Light Grey	Central always

Note: In all cases if the element is a magnet and the appropriate strength is zero, it is shown as a grey line.

5.4 Plot Interactivity

With the adition of the machine axes, some extra interactivity is included to the matplotlib figures.

- zooming - if the 'right-click and drag' zoom feature is used on the machine diagram, the graph will automatically update and follow the machine diagram.
- xlim - setting the horizontal graph limits with the 'xlim' command will update both the machine diagram and the graph.
- querying - right-clicking anywhere on the graph will print out the name of the nearest element in the terminal.

CHAPTER

SIX

MODEL PREPARATION

pymadx contains a series of classes that can be used to programmatically construct a MADX model. The main class is *pymadx.Builder.Machine*.:

```
a = pymadx.Builder.Machine()
a.AddDrift('drift1',1.3)
a.AddQuadrupole('qf1',0.2,1.3454)
a.Write('lattice1')
```

The functions available are documented in *pymadx.Builder module*, but can also easily be found with the built in documentation:

```
a = pymadx.Builder.Machine()
a <tab>
```

to see the list of available functions. Each has a short description and signature that can be viewed with a question mark.:

```
a = pymadx.Builder.Machine()
a.AddQuadrupole?
Signature: a.AddQuadrupole(name='qd', length=0.1, k1=0.0, **kwargs)
Docstring: <no docstring>
File:      ~/physics/reps/pymadx/pymadx/Builder.py
Type:      instancemethod
```

Aside from the lattice elements available, a *pymadx.Beam.Beam* instance can be associated with the machine.:

```
b = pymadx.Beam.Beam()
a.AddBeam(b)
```

CHAPTER

SEVEN

CONVERSION

pymadx provides the ability to convert a MADX model into another format. These are detailed in the sections below.

For conversion of MADX to BDSIM GMAD format, please see the pybdsim documentation http://www.pp.rhul.ac.uk/bdsim/pybdsim/convert.html#madxtfs2gmad.

7.1 Mad8ToMadx

A MAD8 model can be converted to MADX. This relies on the pymad8 package and the conversion is performed there.

7.2 TfsToPtc

A MADX model as described by a full twiss table in a TFS file can be prepared into a new MADX / PTC model suitable for tracking with PTC immediately.

CHAPTER EIGHT

MODULE CONTENTS

This documentation is automatically generated by scanning all the source code. Parts may be incomplete.

pymadx - Royal Holloway utility to manipulate MADX data and models.

Authors:

- Laurie Nevay
- Andrey Abramov
- Stewart Boogert
- William Shields
- Jochem Snuverink
- Stuart Walker

8.1 pymadx.Beam module

Class to create a MADX beam definition.

class `pymadx.Beam.`**`Beam`**(*particletype='e-'*, *energy=1.0*, *distrtype='reference'*, **args*, ***kwargs*)

Bases: `dict`

A class that extends a dictionary for the specific parameters in a MADX beam definition. This class can return a string representation of itself that is valid MADX syntax.

Setter methods are dynamically added based on the distribution selected.

`GetItemStr`(*key*)

`ReturnBeamString`()

`ReturnPtcString`()

`ReturnPtcTwissString`(*basefilename='output'*)

`ReturnTwissString`(*basefilename='output'*)

`SetDistributionType`(*distrtype='reference'*)

`SetEnergy`(*energy=1.0*, *unitsstring='GeV'*)

SetParticleType(*particletype='e-'*)

SetT0(*t0=0.0*, *unitsstring='s'*)

SetX0(*x0=0.0*)

SetXP0(*xp0=0.0*)

SetY0(*y0=0.0*)

SetYP0(*yp0=0.0*)

WriteToFile(*filename*)

8.2 pymadx.Builder module

Builder

Classes for programmatically constructing and writing out a MADX lattice. You can create a lattice using one of the predefined simple lattices or by instantiating the Machine class and adding many elements to it using its various Add methods. This instance may be written out to a MADX input text file using the WriteMachine method.

Classes:

Element - beam line element that always has name and type Line - a list of elements Machine - a sequence of elements and associated options and beam etc.

class `pymadx.Builder.`**Element**(*name*, *category*, ***kwargs*)

Bases: `dict`

Element - a beam element class - inherits dict

Element(name,type,**kwargs)

A beam line element must ALWAYs have a name, and type. The keyword arguments are specific to the type and are up to the user to specify.

Numbers are converted to a python Decimal type to provide higher accuracy in the representation of numbers - 15 decimal places are used.

keysextra()

class `pymadx.Builder.`**Line**(*name*, **args*)

Bases: `list`

DefineConstituentElements()

class `pymadx.Builder.`**Machine**(*verbose=False*)

Bases: `object`

AddBeam(*beam=None*)

AddDecapole(*name='dd'*, *length=0.1*, *k4=0.0*, ***kwargs*)

AddDipole(*category='sbend'*)

category - 'sbend' or 'rbend' - sector or rectangular bend

AddDrift(*name='dr'*, *length=0.1*, ***kwargs*)

AddHKicker(*name='hk'*, *hkick=0*, *length=0*, ***kwargs*)

AddMarker(*name='mk'*, ***kwargs*)

AddMultipole(*name='mp'*, *knl=0*, *ksl=0*, ***kwargs*)

AddOctupole(*name='oc'*, *length=0.1*, *k3=0.0*, ***kwargs*)

AddOptions(**args*, ***kwargs*)

AddPTCTrackAperture(*aperture=[]*)

Add a PTC 6D max aperture for ptc_track command.

AddQuadrupole(*name='qd'*, *length=0.1*, *k1=0.0*, ***kwargs*)

AddSampler(**elementnames*)

AddSextupole(*name='sd'*, *length=0.1*, *k2=0.0*, ***kwargs*)

AddSolenoid(*name='sl'*, *length=0.1*, *ks=0.0*, ***kwargs*)

AddTKicker(*name='tk'*, *vkick=0*, *hkick=0*, *length=0*, ***kwargs*)

AddVKicker(*name='vk'*, *vkick=0*, *length=0*, ***kwargs*)

Append(*object*)

Write(*filename*, *verbose=False*)

Write the machine to a series of gmad files.

next()

class pymadx.Builder.**Sampler**(*name*)

Bases: `object`

Class that can return the appropriate sampler syntax if required.

pymadx.Builder.**WriteMachine**(*machine(machine)*, *filename(string)*, *verbose(bool)*)

Write a lattice to disk. This writes several files to make the machine, namely:

- filename_components.madx - component files (max 10k per file)
- filename_sequence.madx - lattice definition
- filename_samplers.madx - sampler definitions (max 10k per file)
- **filename.gmad - suitable main file with all sub**
  files in correct order

These are prefixed with the specified filename / path.

8.3 pymadx.Data module

Classes to load and manipulate data from MADX.

class pymadx.Data.**Aperture**(*filename=None*, *verbose=False*)

Bases: *Tfs*

Inherits Tfs, with added functionality specific to apertures: changing aperture types, getting the aperture at a specific S, etc.

Keyword Arguments: filename – TFS file to be loaded (default None) verbose – (default False) beamLossPattern – Whether to apply beamLossPattern's interpretation of APER numbers to infer APERTYPE (default False)

CheckKnownApertureTypes()

GetApertureAtS(*sposition*)

Return a dictionary of the aperture information specified at the closest S position to that requested - may be before or after that point.

GetEntriesBelow(*value=8*, *keys='all'*)

GetExtent(*name*)

Get the x and y maximum +ve extent (assumed symmetric) for a given entry by name.

GetExtentAll()

Get the x and y maximum +ve extent (assumed symmetric) for the full aperture instance.

returns x,y where x and y are 1D numpy arrays

GetExtentAtS(*sposition*)

Get the x and y maximum +ve extent (assumed symmetric) for a given s position.

GetNonZeroItems()

Return a copy of this class with all non-zero items removed.

Plot(*machine=None*, *outputfilename=None*, *plot='xy'*, *plotapertype=True*)

This plots the aperture extent in x and y.

This replaces the base class Tfs Plot method.

Inputs:

title (str) - The title of the resultant plot (default: None) outputfilename (str) - Name without extension of the output file if desired (default: None) machine (str or pymadx.Data.Tfs) - TFS file or TFS istance to plot a machine lattice from (default: None) plot (str) - Indicates whcih aperture to plot - 'x' for X, 'y' for Y and 'xy' for both (default: 'xy') plotapertype (bool) - If enabled plots the aperture type at every definted aperture point as a color-coded dot (default: False)

PlotN1(*machine=None*, *outputfilename=None*)

Plot the N1 aperture value from MADX.

Requires N1 and S column.

Optional "machine" argument is string to or pymadx.Data.Tfs instance for twiss description to provide a machine diagram on top.

RemoveAboveValue(*limits=8*, *keys='all'*)

RemoveBelowValue(*limits*, *keys='all'*)

Return a copy of the aperture instance with all entries where any of the aperture values are below value. The default is the tolerance as defined by SetZeroTolerance().

RemoveDuplicateSPositions()

Takes the first aperture value for entries with degenerate S positions and removes the others.

RemoveNoApertureTypeEntries()

Return a copy of this instance with any null aperture types removed.

Aperture type of “” and “NONE” will be removed.

ReplaceType(*existingType*, *replacementType*)

SetZeroTolerance(*tolerance*)

Set the value below which aperture values are considered 0.

pymadx.Data.**CheckItsTfs**(*tfsfile*)

Ensure the provided file is a Tfs instance. If it’s a string, ie path to a tfs file, open it and return the Tfs instance.

tfsfile can be either a tfs instance or a string.

pymadx.Data.**CheckItsTfsAperture**(*tfsfile*)

Ensure the provided file is an Aperture instance. If it’s a string, ie path to a tfs file, open it and return the Tfs instance.

tfsfile can be either a tfs instance or a string.

pymadx.Data.**GetApertureExtent**(*aper1*, *aper2*, *aper3*, *aper4*, *aper_type*)

Get the maximum +ve half extent in x and y for a given aperture model and (up to) four aperture parameters.

returns x,y

pymadx.Data.**PrintMADXApertureTypes**()

class pymadx.Data.**Tfs**(*filename=None*, *verbose=False*)

Bases: `object`

MADX Tfs file reader

```
>>> a = Tfs()
>>> a.Load('myfile.tfs')
>>> a.Load('myfile.tar.gz') -> extracts from tar file
```

or

```
>>> a = Tfs("myfile.tfs")
>>> b = Tfs("myfile.tar.gz")
```

a has data members:
header - dictionary of header items
columns - list of column names
formats - list of format strings for each column
data - dictionary of entries in tfs file by name string
sequence - list of names in the order they appear in the file
nitems - number of items in sequence

NOTE: if no column “NAME” is found, integer indices are used instead

See the various methods inside a to get different bits of information:

```
>>> a.ReportPopulations?
```

Examples:

```
>>> a['IP.1']  # returns dict for element named "IP.1"
>>> a[:30]     # returns list of dicts for elements up to number 30
>>> a[345]     # returns dict for element number 345 in sequence
```

Clear()

Empties all data structures in this instance.

ColumnIndex(*columnstring*)

Return the index to the column matching the name

REMEMBER: excludes the first column NAME 0 counting

ComponentPerturbs(*indexInSequence*)

Returns names of variables which would perturb a particle. Some components written out in TFS are redundant, so it's useful to know which components perturb a particle's motion. This is likely not an exhaustive check so refer to source if unsure.

Checks integrated stengths (but not if L=0), HKICK and VKICK

indexInSequence - index of component to be checked. terse - print out the parameters which perturb if False

ConcatenateMachine(**tfs*)

This is used to concatenate machines.

EditComponent(*index*, *variable*, *value*)

Edits variable of component at index and sets it to value. Can only take indices as every single element in the sequence has a unique definition, and components which may appear degenerate/reused are in fact not in this data model.

ElementPerturbs(*component*)

Search an invidivual dictionary representing a row in the TFS file for as to whether it perturbs.

GetCollimators()

Returns a Tfs instance containing any type of collimator (including COLLLIMATOR, RCOLLIMATOR and ECOLLIMATOR).

GetColumn(*columnstring*)

Return a numpy array of the values in columnstring in order as they appear in the beamline

GetColumnDict(*columnstring*)

return all data from one column in a dictionary

note not in order

GetElementNamesOfType(*typename*)

Returns a list of the names of elements of a certain type. Typename can be a single string or a tuple or list of strings.

Examples:

```
>>> GetElementsOfType('SBEND')
>>> GetElementsOfType(['SBEND','RBEND'])
>>> GetElementsOfType(('SBEND','RBEND','QUADRUPOLE'))
```

GetElementsOfType(*typename*)

Returns a Tfs instance containing only the elements of a certain type. Typename can be a sintlge string or a tuple or list of strings.

This returns a Tfs instance with all the same capabilities as this one.

GetElementsWithTextInName(*text*)

Returns a Tfs instance containing only the elements with the string in text in the their name.

This returns a Tfs instance with all the same capabilities as this one.

GetRow(*elementname*)

Return all data from one row as a list

GetRowDict(*elementname*)

Return a dictionary of all parameters for a specifc element given by element name.

note not in order

GetSegment(*segmentnumber*)

IndexFromGmadName(*gmadname*, *verbose=False*)

Returns the indices of elements which match the supplied gmad name. Useful because tfs2gmad strips punctuation from the component names, and irritating otherwise to work back. When multiple elements of the name match, returns the indices of all the components in a list. Arguments: gmadname : The gmad name of a component to search for. verbose : prints out matching name indices and S locations. . Useful for discriminating between identical names.

IndexFromName(*namestring*)

Return the index of the element named namestring. Raises ValueError if not found.

IndexFromNearestS(*S*)

return the index of the beamline element which CONTAINS the position S.

Note: For small values beyond smax, the index returned will be -1 (i.e. the last element).

InterrogateItem(*itemname*)

Print out all the parameters and their names for a particlular element in the sequence identified by name.

Load(*filename*, *verbose=False*)

```
>>> a = Tfs()
>>> a.Load('filename.tfs')
```

Read the tfs file and prepare data structures. If 'tar' or 'gz are in the filename, the file will be opened still compressed.

NameFromIndex(*integerindex*)

return the name of the beamline element at index

NameFromNearestS(*S*)

return the name of the beamline element clostest to S

Plot(*title=''*, *outputfilename=None*, *machine=True*, *dispersion=False*, *squareroot=True*)

Plot the Beta amplitude functions from the file if they exist.

squareroot -> whether to square root the beta functions or not (default = True)

PlotCentroids(*title=''*, *outputfilename=None*, *machine=True*)

Plot the centroid in the horizontal and vertical from the file if they exist.

PlotSigma(*title=''*, *outputfilename=None*, *machine=True*, *dispersion=False*)

Plot the beam size.

PrintBasicBeamProperties(*elementname*)

Print centroid, transverse momentum, beta, alpha and sigma in x and y.

Will fail if these are not available.

RenameElement(*index*, *new*)

Rename indexed element.

ReportPopulations()

Print out all the population of each type of element in the beam line (sequence)

SplitElement(*SSplit*)

Splits the element found at SSplit given, performs the necessary operations on the lattice to leave the model functionally identical and returns the indices of the first and second component. Element new name will be the same as the original except appended with a number corresponding to its location in the list of previously identically defined components used in the sequence and either "split_1" or "split_2" depending on which side of the split it is located. It is necessary to append both of these numbers to ensure robust name mangling.

WARNING: DO NOT SPLIT THE ELEMENT WHICH MARKS THE BEGINNING OF YOUR LATTICE. YOUR OPTICS WILL BE WRONG!

Write(*outputfilename*, *columns=None*, *removePymadxColumns=True*)

Write this instance to file in MADX TFS format.

Specify column names with columns=['S', 'L'] for example.

next()

8.4 pymadx.Plot module

Ploting script for madx TFS files using the pymadx Tfs class

pymadx.Plot.**AddMachineLatticeToFigure**(*figure*, *tfsfile*, *tightLayout=True*, *reverse=False*, *offset=None*)

Add a diagram above the current graph in the figure that represents the accelerator based on a madx twiss file in tfs format.

Note you can use matplotlib's gcf() 'get current figure' as an argument.

```
>>> pymadx.Plot.AddMachineLatticeToFigure(gcf(), 'afile.tfs')
```

A pymadx.Tfs class instance or a string specifying a tfs file can be supplied as the second argument interchangeably.

If the reverse flag is used, the lattice is plotted in reverse only. The tfs instance doesn't change.

Offset can optionally be the name of an object in the lattice (exact name match).

If both offset and reverse are used, reverse happens first. The right click searching works with the reverse and offset similarly.

`pymadx.Plot.`**`Aperture`**`(`*aperture*, *machine=None*, *outputfilename=None*, *plot='xy'*, *plotapertype=True*`)`

Plots the aperture extents vs. S from a pymadx.Data.Aperture instance.

Inputs:

aperture (pymadx.Data.Aperture) - the aperture model to plot from machine (str or pymadx.Data.Tfs) - TFS file or TFS istance to plot a machine lattice from (default: None) outputfilename (str) - Name without extension of the output file if desired (default: None) plot (str) - Indicates whcih aperture to plot - 'x' for X, 'y' for Y and 'xy' for both (default: 'xy') plotapertype (bool) - If enabled plots the aperture type at every definted aperture point as a color-coded dot (default: False)

`pymadx.Plot.`**`ApertureN1`**`(`*aperture*, *machine=None*, *outputfilename=None*`)`

Plot the N1 aperture value from MADX.

Requires N1 and S column.

Optional "machine" argument is string to or pymadx.Data.Tfs instance for twiss description to provide a machine diagram on top.

`pymadx.Plot.`**`Beta`**`(`*tfsfile*, *title=''*, *outputfilename=None*, *machine=True*, *dispersion=False*, *squareroot=True*`)`

Plot sqrt(beta x,y) as a function of S. By default, a machine diagram is shown at the top of the plot.

Optionally set dispersion=True to plot x dispersion as second axis. Optionally turn off machine overlay at top with machine=False Specify outputfilename (without extension) to save the plot as both pdf and png.

`pymadx.Plot.`**`Centroids`**`(`*tfsfile*, *title=''*, *outputfilename=None*, *machine=True*`)`

Plot the centroid (mean) x and y from the a Tfs file or pymadx.Tfs instance.

tfsfile - can be either a string or a pymadx.Tfs instance. title - optional title for plot outputfilename - optional name to save file to (extension determines format) machine - if True (default) add machine diagram to top of plot

`pymadx.Plot.`**`MachineDiagram`**`(`*tfsfile*, *title=None*, *reverse=False*`)`

Plot just a machine diagram on its own. The S axis is shown.

Parameters

- **`tfsfile`** (*`pymadx.Data.TFS, str`*) – TFS instance or file name of lattice to plot.
- **`title`** (*`None, str`*) – Title for plot.
- **`reverse`** (*`bool`*) – Whether to reverse the direction of the machine.

`pymadx.Plot.`**`Sigma`**`(`*tfsfile*, *title=''*, *outputfilename=None*, *machine=True*, *dispersion=False*`)`

Plot sqrt(beta x,y) as a function of S. By default, a machine diagram is shown at the top of the plot.

Optionally set dispersion=True to plot x dispersion as second axis. Optionally turn off machine overlay at top with machine=False Specify outputfilename (without extension) to save the plot as both pdf and png.

`pymadx.Plot.`**`Survey`**`(`*tfsfile*, *title=''*, *outputfilename=None*`)`

Plot the x and z coordinates from a tfs file.

`pymadx.Plot.`**`TwoMachineDiagrams`**`(`*tfsTop*, *tfsBottom*, *labelTop=None*, *labelBottom=None*, *title=None*, *reverse=False*`)`

Plot just a machine diagram on its own. The S axis is shown.

Parameters

- **`tfsfile`** (*`pymadx.Data.TFS, str`*) – TFS instance or file name of lattice to plot.
- **`title`** (*`None, str`*) – Title for plot.
- **`reverse`** (*`bool`*) – Whether to reverse the direction of the machine.

8.5 pymadx.Ptc module

Classes to handle PTC runs and data.

class pymadx.Ptc.**FlatGenerator**(*mux=0.0*, *widthx=0.001*, *mupx=0.0*, *widthpx=0.001*, *muy=0.0*, *widthy=0.001*, *mupy=0.0*, *widthpy=0.001*)

Bases: `object`

Simple ptc inray file generator - even distribution

Generate(*nToGenerate=100*, *fileName='inrays.madx'*)

returns an Inrays structure

class pymadx.Ptc.**GaussGenerator**(*gemx=1e-10*, *betax=0.1*, *alfx=0.0*, *gemy=1e-10*, *betay=0.1*, *alfy=0.0*, *sigmat=1e-12*, *sigmapt=1e-12*)

Bases: `object`

Simple ptx inray file generator

Generate(*nToGenerate=1000*, *fileName='inrays.madx'*)

returns an Inrays structure

class pymadx.Ptc.**Inray**(*x=0.0*, *px=0.0*, *y=0.0*, *py=0.0*, *t=0.0*, *pt=0.0*)

Bases: `object`

Class for a madx ptc input ray x : horizontal position [m] px : horizontal canonical momentum p_x/p_0 y : vertical position [m] py : vertical canonical momentum p_y/p_0 t : c*(t-t0) [m] pt : (delta-E)/(pc)

use str(Inray) to get the representation for file writing

class pymadx.Ptc.**Inrays**

Bases: `list`

Class based on python list for Inray class

AddParticle(*x=0.0*, *px=0.0*, *y=0.0*, *py=0.0*, *t=0.0*, *pt=0.0*)

Clear()

Plot()

Statistics()

Write(*filename*)

pymadx.Ptc.**LoadInrays**(*fileName*)

Load input rays from file fileName : inrays.madx return : Inrays instance

pymadx.Ptc.**PlotInrays**(*i*)

Plot Inrays instance, if input is a sting the instance is created from the file

pymadx.Ptc.**WriteInrays**(*fileName*, *inrays*)

8.6 pymadx.PtcAnalysis module

Analysis utilities for PTC data.

class pymadx.PtcAnalysis.**PtcAnalysis**(*ptcInput=None*, *ptcOutput=None*)

Bases: `object`

Deprecated.

Optical function calculation for PTC tracking data.

This has be reimplemented and replaced by C++ implementation in rebdsim.

CalculateOpticalFunctions(*output*)

Calulates optical functions from a PTC output file

output - the name of the output file

SamplerLoop()

CHAPTER

NINE

VERSION HISTORY

9.1 v 2.0.0 - 2023 / 03 / 16

- Move to Python 3 entirely. Require at least Python 3.7.
- Package layout and build system changed to more modern declarative package. All source code is now in `pymadx/src/pymadx`. The version number throughout the code is dynamically generated from the git tag.
- Added plot for 1 or 2 machine diagrams only.
- Fix aperture plots due to typo in code.
- Fix string type comparison for modern Python (i.e. don't use numpy internal alises).

9.2 v 1.8.2 - 2021 / 06 / 16

- Fix for plot name filtering.
- Tweaked orange for solenoids.

9.3 v 1.8.1 - 2020 / 12 / 16

- Fix for step size in Tfs slicing.
- More tolerant plotting for machine diagrams with just keyword, S and L as colums (ignoring K1L).
- Ensure machine diagram x limit is full machine length by default.

9.4 v 1.8.0 - 2019 / 06 / 08

9.4.1 New Features

- Switch to Python 3. Should be Python 2.7 compatible.
- Venv support in Makefile thanks to Kyrre Ness Sjoebaek.
- Ability to write out a Tfs instance permitting comlete loading, editing and writing.
- Plus operator for Tfs instances to add them together.

9.4.2 Bug Fixes

- Use exact Hamiltonian for PTC jobs prepared from pymadx as we commonly use it to compare larger amplitude particle tracking where the approximate Hamiltonian can be quite wrong.
- Tolerate minimal aperture columns. i.e. only APER_1. Have to do this as there's no standard in writing out apertures and everyone picks their own with missing bits of information.

9.5 v 1.7.1 - 2019 / 04 / 20

9.5.1 Bug Fixes

- Fix Data.Aperture.RemoveBelowValue logic, which also applies to GetNonZeroItems.
- Tolerate no pytransport at import.

9.6 v 1.7 - 2019 / 02 / 27

9.6.1 New Features

- Return PTC beam definition from the Beam class.
- Print basic beam summary from TFS file for given element.
- Ability to split an element loaded from a TFS file correctly.

9.6.2 General

- Update copyright for 2019.

9.7 v 1.6 - 2018 / 12 / 12

9.7.1 General

- Reimplemented machine diagram drawing to be more efficient when zooming and fix zordering so bends and then quadrupoles are always on top.
- Dispersion optional for optics plotting.
- H1 and H2 now passed through conversion of MADX TFS to PTC input format.
- Solenoid added to MADX TFS to PTC converter.
- Revised bend conversion for MADX TFS to PTC converter.

9.8 v 1.5 - 2018 / 08 / 24

9.8.1 New Features

- Support for tkicker.
- Support for kickers in MADX to PTC.

9.8.2 General

- Improved aperture handling.

9.8.3 Bug Fixes

- Several bugs in Aperture class fixed.

9.9 v 1.4 - 2018 / 06 / 23

9.9.1 New Features

- Support of just gzipped files as well as tar gzipped.

9.9.2 General

- Improved SixTrack aperture handling.

9.10 v 1.2 - 2018 / 05 / 23

9.10.1 New Features

- Write a beam class instance to a separate file.
- Add ptc_track maximum aperture to a model.
- Concatenate TFS instances.
- N1 aperture plot as well as physical aperture plot.
- Output file naming for plots for MADX MADX comparison.
- MADX Transport comparison plots.

9.10.2 General

- Changes to some plot arguments.
- 'Plot' removed from plot functions name as redundant.
- Transport conversion moved to pytransport.

9.10.3 Bug Fixes

- Machine plot now deals with 'COLLIMATOR' type correctly.

9.11 v 1.1 - 2018 / 04 / 10

9.11.1 New Features

- Improved options for writing PTC job for accurate comparison.
- Support for subrelativistic machines - correct MADX definition of dispersion.
- Plots for beam size including dispersion.
- MADX MADX Twiss comparison plots.

9.11.2 Bug Fixes

- Removal of reverse slicing as it didn't work and is very difficult to support as MADX typically returns optical functions at the end of an element. Some columns however are element specific (such as L).
- Fixed exception catching.
- Fix beam size for subrelativistic machines. MADX really provides Dx/Beta.
- Fix index searching from S location.
- Fix PTC analysis.
- Fix conversion to PTC for fringe fields.

9.12 v 1.0 - 2017 / 12 / 05

9.12.1 New Features

- GPL3 licence introduced.
- Compatability with PIP install system.
- Manual.
- Testing suite.

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

I

K

L

M

N

P

R

S

T

W