pybdsim Documentation

Release 2.4.0

Royal Holloway

Jun 15, 2021

CONTENTS

pybdsim is a Python package to aid in the preparation, running and validation of BDSIM models.

CHAPTER

ONE

LICENCE & DISCLAIMER

CHAPTER TWO

AUTHORSHIP

The following people have contributed to pybdsim:

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

CHAPTER

THREE

INSTALLATION

3.1 Requirements

- pybdsim is developed exclusively for Python 2.7.
- Matplotlib
- Numpy
- Scipy
- fortranformat
- root-numpy
- ROOT Python interface
- pip

3.2 Installation

To install pybdsim, simply run `make install` from the root pybdsim directory.:

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

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

CHAPTER

FOUR

BUILDING MODELS

pybdsim provides a series of classes that allows a BDSIM model to be built programmatically in Python and finally written out to BDSIM input syntax ('gmad').

4.1 Creating A Model

The `Machine` class provides the functionality to create a BDSIM model. This would instantiated and a sequence is defined by adding accelerator elements in order to that instance by calling functions such as `AddDipole()`. Extra information can then be associated with that Machine instance and finally, it can be written out to a series of gmad files as input to BDSIM. For example:

```
>>> a = pybdsim.Builder.Machine()
>>> a.AddDrift()
```

The arguments can generally be found by using a question mark on a function.:

```
>>> a.AddDrift?
Signature: a.AddDrift(name='dr', length=0.1, **kwargs)
Docstring: Add a drift to the beam line
File:      ~/physics/reps/pybdsim/Builder.py
Type:      instancemethod
```

4.2 Adding Options

No options are required to run the most basic BDSIM model. However, it is often advantageous to specify at leat a few options such as the physics list and default aperture. To add options programmatically, there is an options class. This is instantiated and then 'setter' methods are used to set values of parameters. This options instance can then be assocated with a machine instance. For example:

```
>>> o = pybdsim.Options.Options()
>>> o.SetPhysicsList('em hadronic decay muon hadronic_elastic')

>>> a = pybdsim.Builder.Machine()
>>> a.AddOptions(o)
```

The possible options can be seen by using tab complete in ipython:

```
>>> a.Set<tab>
```

Note: Only the most common options are currently implement. Please see *Feature Request* to request others.

4.3 Adding a Beam

A beam definition that specifies at least the particle type and total energy is required to run a BDSIM model. The machine class will provide a default such that the model will run 'out of the box', but is of course of interest to specify these options. To add a beam definition, there is a beam class. This is instantiated and then 'setter' methods are used to set values of parameters. this beam instance can then be associated with a machine instance. For example:

```
>>> b = pybdsim.Beam.Beam()
>>> b.SetDistributionType('reference')
>>> b.SetEnergy(25, 'GeV')
>>> b.SetParticleType('proton')

>>> a = pybdsim.Builder.Machine()
>>> a.AddBeam(b)
```

Note: More setter functions will dynamically appear based on the distribution type set.

4.4 Writing a Machine

Once completed, a machine can be written out to gmad files to be used as input for BDSIM. This is done as follows:

```
>>> a = pybdsim.Builder.Machine()
>>> a.Write('outputfilename')
```

4.5 Units

The user may supply units as strings that will be written to the gmad syntax as a Python tuple. For example:

```
>>> a = pybdsim.Builder.Machine()
>>> a.AddDrift('d1', (3.2, 'm'))
```

This will result in the following gmad syntax:

```
>>> print a[0]
d1: drift, l=3.2*m;
```

Note: There is no checking on the string supplied, so it is the users responsibility to supply a valid unit string that BDSIM will accept.

4.6 kwargs - Flexibility

'kwargs' are optional keyword arguments in Python. This allows the user to supply arbitrary options to a function that can be instspected inside the function as a dictionary. BDSIM gmad syntax to define an element generally follows the pattern:

```
name : type, parameter1=value, parameter2=value;
```

Many parameters can be added and this syntax is regularly extended. It would therefore be impractical to have every function with all the possible arguments. To solve this problem, the `**kwargs` argument allows the user to specify any option that will be passed along and written to file in the element definition as 'key=value'. For example:

```
>>> a = pybdsim.Builder.Machine()
>>> a.AddDrift('drift321', 3.2, aper1=5, aper2=4.5, apertureType="rectangular")
```

This will result in the following gmad syntax being written:

```
>>> print a[0]
drift321: drift, apertureType="rectangular", aper2=4.5, aper1=5, l=3.2;
```

Anywhere you see a function with the last argument as `**kwargs`, this feature can be used.

The arguments included in the function signatures are the minimum arguments required for functionality.

CHAPTER FIVE

CONVERTING MODELS

pybdsim provdies converters to allow BDSIM models to prepared from optical descriptions of accelerators in other formats such as MADX and MAD8.

The following converters are provided and described here:

- MADX to BDSIM
  - *MadxTfs2Gmad*
  - *MadxTfs2GmadStrength*
- MAD8 to BDSIM
  - *Mad8Twiss2Gmad (using saved TWISS output)*
- Transport to BDSIM
  - *pytransport*
- BDSIM Primary Particle Conversion
  - *BDSIM Primaries To Others*

5.1 MadxTfs2Gmad

A MADX lattice can be easily converted to a BDSIM gmad input file using the supplied python utilities. This is achieved by

1. preparing a tfs file with madx containing all twiss table information
2. converting the tfs file to gmad using pybdsim

5.1.1 Preparing a Tfs File

The twiss file can be prepared by appending the following MADX syntax to the end of your MADX script:

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

where *SEQUENCENAME* is the name of the sequence in madx. By not specifying the output columns, a very large file is produced containing all possible columns. This is required to successfully convert the lattice. If the tfs file contains insufficient information, pybdsim will not be able to convert the model.

Note: The python utilities require "*.tfs*" suffix as the file type to work properly.

5.1.2 Converting the Tfs File

Once prepared, the Tfs file can be converted. The converter is used as follows:

```
>>> pybdsim.Convert.MadxTfs2Gmad('inputfile.tfs', 'latticev1')
```

The conversion returns three objects, which are the `pybdsim.Builder.Machine` instance as converted, a second *Machine* that isn't split by aperture and a list of any ommitted items by name.

```
>>> a,b,c = pybdsim.Convert.MadxTfs2Gmad('inputfile.tfs', 'latticev1')
```

where *latticev1* is the output name of the converted model. The converter has the ability to split items in the original TFS file if an aperture is specified somewhere inside that element - use for disjoint aperture definitions. If a directory is used in the output name, this will be created automatically, for example:

```
>>> a,o = pybdsim.Convert.MadxTfs2Gmad('inputfile.tfs', 'test/latticev1')
```

will create a directory *test* if it doesn't exist already.

There are a few options that provide useful functionality for conversion:

tfs	path to the input tfs file or pymadx.Data.Tfs instance
outputfilename	requested output file
startname	the name (exact string match) of the lattice element to start the machine at this can also be an integer index of the element sequence number in madx tfs.
stopname	the name (exact string match) of the lattice element to stop the machine at this can also be an integer index of the element sequence number in madx tfs.
stepsize	the slice step size. Default is 1, but -1 also useful for reversed line.
ignorezerolengthitems	nothing can be zero length in bdsim as real objects of course have some finite size. Markers, etc are acceptable but for large lattices this can slow things down. True allows to ignore these altogether, which doesn't affect the length of the machine.
samplers	can specify where to set samplers - options are None, 'all', or a list of names of elements (normal python list of strings). Note default 'all' will generate separate outputfilename_samplers.gmad with all the samplers which will be included in the main .gmad file - you can comment out the include to therefore exclude all samplers and retain the samplers file.
aperturedict	Aperture information. Can either be a dictionary of dictionaries with the the first key the exact name of the element and the daughter dictionary containing the relevant bdsim parameters as keys (must be valid bdsim syntax). Alternatively, this can be a pymadx.Aperture instance that will be queried.
aperlocalpositions	Dictionary of element indices to a list of pairs of the form (local_point, aperdict), for example (0.1, {"APER1": "CIRCULAR", "APER1": 0.4}). This kwarg is mutually exclusive with "aperturedict".
collimatordict	A dictionary of dictionaries with collimator information keys should be exact string match of element name in tfs file value should be dictionary with the following keys: "bdsim_material" - the material "angle" - rotation angle of collimator in radians "xsize" - x full width in metres "ysize" - y full width in metres
userdict	A python dictionary the user can supply with any additional information for that particular element. The dictionary should have keys matching the exact element name in the Tfs file and contain a dictionary itself with key, value pairs of parameters and values to be added to that particular element.
verbose	Print out lots of information when building the model.
beam	True | False - generate an input gauss Twiss beam based on the values of the twiss parameters at the beginning of the lattice (startname) NOTE - we thoroughly recommend checking these parameters and this functionality is only for partial convenience to have a model that works straight away.
flipmagnets	True | False - flip the sign of all k values for magnets - MADX currently tracks particles agnostic of the particle charge - BDSIM however, follows the definition strictly - positive k -> horizontal focussing for positive particles therefore, positive k -> vertical focussing for negative particles. Use this flag to flip the sign of all magnets.
usemadxaperture	True | False - use the aperture information in the TFS file if APER_1 and APER_2 columns exist. Will only set if they're non-zero. Supercedes kwargs *aperturedict* and *aperlocalpositions*.
defaultAperture	The default aperture model to assume if none is specified.
biases	Optional list of bias objects to be defined in own _bias.gmad file. These can then be attached either with allelementdict for all components or userdict for individual ones.
allelementdict	Dictionary of parameter/value pairs to be written to all components.
optionsDict	Optional dictionary of general options to be written to the bdsim model options.
linear	Only linear optical components
overwrite	Do not append an integer to the base file name if it already exists. Instead overwrite the files.
allNamesUnique	Treat every row in the TFS file/instance as a unique element. This makes it easier to edit individual components as they are guaranteed to appear only once in the entire resulting GMAD lattice.

The user may convert only part of the input model by specifying *startname* and *stopname*.

Generally speaking, extra information can be folded into the conversion via a user supplied dictionary with extra parameters for a particular element by name. For a given element, for example 'drift123', extra parameters can be speficied in a dictionary. This leads to a dictionary of dictionaries being supplied. This is a relatively simple structure the user may prepare from their own input format and converters in Python. For example:

```
>>> drift123dict = {'aper1':0.03, 'aper2':0.05, 'apertureType':'rectangular'}
>>> quaddict = {'magnetGeometryType':'polesfacetcrop}
>>> d = {'drift123':drift123dict, 'qf1x':quaddict}
>>> a,o = pybdsim.Convert.MadxTfs2Gmad('inputfile.tfs', 'latticev1', userdict=d)
```

5.1.3 Notes

1) The name must match the name given in the MADX file exactly.
2) Specific arguments may be given for aperture (*aperturedict*), or for collimation (*collimatordict*), which are used specifically for those purposes.
3) There are quite a few options and these are described in *pybdsim.Convert*.
4) The BDSIM-provided pymadx package is required for this conversion to work.
5) The converter will alter the names to remove forbidden characters in names in BDSIM such as '$' or '!'.

5.1.4 Preparation of a Small Section

For large accelerators, it is often required to model only a small part of the machine. We recommend generating a Tfs file for the full lattice by default and trimming as required. The pymadx.Data.Tfs class provides an easy interface for trimming lattices. The first argument to the pybdsim.Convert.MadxTfs2Gmad function can be either a string describing the file location or a pymadx.Data.Tfs instance. The following example trims a lattice to only the first 100 elements:

```
>>> a = pymadx.Data.Tfs("twiss_v5.2.tfs")
>>> b = a[:100]
>>> m,o = pybdsim.Convert.MadxTfs2Gmad(b, 'v5.2a')
```

5.2 MadxTfs2GmadStrength

This is a utility to prepare a strength file file from a Tfs file. The output gmad file may then be included in an existing BDSIM gmad model after the lattice definition which will update the strengths of all the magnets.

5.3 Mad8Twiss2Gmad (using saved TWISS output)

Note: This requires the https://bitbucket.org/jairhul/pymad8 package.

A MAD8 lattice can be easily converted to a BDSIM gmad input file using the supplied python utilities. This is achieved by

1. preparing twiss, envel, survey and structure tape files with mad8
2. echo variables in the mad8 job log (SIGPT, SIGT)
3. converting the tape files to gmad using pybdsim

5.3.1 Running mad8

The following variables need to be defined in the Mad8 job from a BETA0

```
EMITX     := 0.01e-6
EMITY     := 0.01e-6
BLENG     := 0.3e-3
ESPRD     := 0.1e-3
TALFX     := BETA0[alfx]
TALFY     := BETA0[alfy]
TBETX     := BETA0[betx]
TBETY     := BETA0[bety]
TGAMX     := (1+TALFX*TALFX)/TBETX
TGAMY     := (1+TALFY*TALFY)/TBETY
SIG11     := EMITX*TBETX
SIG21     := -EMITX*TALFX
SIG22     := EMITX*TGAMX
SIG33     := EMITY*TBETY
SIG43     := -EMITY*TALFY
SIG44     := EMITY*TGAMY
C21       := SIG21/SQRT(SIG11*SIG22)
C43       := SIG43/SQRT(SIG33*SIG44)
S0_I1.G1  : SIGMA0, SIGX=SQRT(SIG11), SIGPX=SQRT(SIG22), R21=C21, &
                    SIGY=SQRT(SIG33), SIGPY=SQRT(SIG44), R43=C43, &
                    SIGT=BLENG, SIGPT=ESPRD

VALUE, EMITX
VALUE, EMITY
VALUE, ESPRD
VALUE, BLENG
```

Creating the output files:

```
use, <latticename>
twiss, beta0=BETA0, save, tape=twiss_<latticename> , rtape=rmat_<latticename>
structure, filename=struct_<latticename>
envelope, sigma0=SIGMA0, save=envelope, tape=envel_<latticename>
```

Optionally the following files are required:

```
survey, tape=survey_<latticename>
```

Running mad8:

```
mad8s < <jobfilename> > <jobfilename>.log
```

5.3.2 Converting the Mad8 files

Two steps are required to create the model from the Mad8 files, first to create template files for the collimators and apertures from the Mad8, this is done by running the following commands

```
pybdsim.Convert.Mad8MakeCollimatorTemplate(<inputtwissfilename>,<collimatordbfilename>
↪)
pybdsim.Convert.Mad8MakeApertureTemplate(<inputtwissfilename>,<aperturedbfilename>)
```

Copy the <collimatordbfilename> to `collimator.dat` and <aperturedbfilename> to `apertures.dat` Once prepared, the Tape files can be converted. The converter is used as follows:

```
pybdsim.Convert.Mad8Twiss2Gmad(<inputtwissfilename>,<outputgamdfilename>)
```

5.4 pytransport

https://bitbucket.org/jairhul/pytransport is a separate utility to convert transport models into BDSIM ones.

5.5 BDSIM Primaries To Others

The particle coordinates recorded by BDSIM may be read from an output ROOT file and written to another format. This can be used for example to ensure the exact same coordinates are used in multiple BDSIM simulations, or to when comparing BDSIM to other tracking codes such as PTC. It can also be used for example to pass coordinates from one BDSIM simulation to another where a detailed simulation of a region of the machine may be desired without the need to simulate the preceding section of the machine.

For the conversion to PTC coordinate convension, it is assumed that PTC calculations are performed in 6D, and that the *TIME* flag in the *PTC_CREATE_LAYOUT* routine is false, meaning the fifth and sixth coordinates are $-pathlength$ and $\delta p = \frac{(p-p_0)}{p_0}$ respectively.

For all converters, a *start* number, *n* can be specified which converts from the nth particle onwards. The number of particles converted can be specified with the *ninrays* argument. For example, to convert particles 2 to 10 only, the arguments supplied would be `start=2, ninrays=9`.

5.5.1 BdsimPrimaries2Ptc

The primary BDSIM coordinates are converted to PTC format. The converter is used as follows:

```
>>> pybdsim.Convert.BdsimPrimaries2Ptc('output.root', 'inrays.dat')
```

5.5.2 BdsimSampler2Ptc

The BDSIM coordinates from a provided sampler name are converted to PTC format. The converter is used as follows:

```
>>> pybdsim.Convert.BdsimSampler2Ptc('output.root', 'inrays.dat','DR1')
```

This will convert the coordinates recorded in sampler *DR1*. Only the primary particles are converted.

5.5.3 BdsimPrimaries2BdsimUserFile

The primary BDSIM coordinates are converted to a BDSIM *userFile* format. The converter is used as follows:

```
>>> pybdsim.Convert.BdsimPrimaries2BdsimUserFile('output.root', 'inrays.dat')
```

5.5.4 BdsimSampler2BdsimUserFile

The BDSIM coordinates from a provided sampler name are converted to a BDSIM *userFile* format. The converter is used as follows:

```
>>> pybdsim.Convert.BdsimSampler2BdsimUserFile('output.root', 'inrays.dat','DR1')
```

The time coordinate recorded in the input file will be finite if the sampler being converted is not at the start of the machine. This function is intended to convert particles into a primary distribution, therefore the time coordinate must be centred around *t=0*. As the nominal time is not recorded, the mean time is subtracted from all particles. Note that at low particle numbers, statistical fluctuations may result in the mean time being significantly different from the nominal time.

5.5.5 BdsimPrimaries2Madx

The primary BDSIM coordinates are converted to madx format. The converter is used as follows:

```
>>> pybdsim.Convert.BdsimPrimaries2Madx('output.root', 'inrays.dat')
```

5.5.6 BdsimPrimaries2Mad8

The primary BDSIM coordinates are converted to mad8 format. The converter is used as follows:

```
>>> pybdsim.Convert.BdsimPrimaries2Mad8('output.root', 'inrays.dat')
```

CHAPTER

SIX

MODEL COMPARISON

Once a BDSIM model has been prepared from another model, it is of interest to validate it to ensure the model has been prepared correctly.

6.1 Preparing Optics with BDSIM

The BDSIM model should be run with a 'core' beam distribution - ie typically a Gaussian or Twiss Gaussian that will match the optics of the lattice. For a physics study one might use a halo, but this is unsuitable for optics validation.

To compare, a BDSIM model is run with samplers attached to each element. This records all of the particle coordinates at the end of each element. Once finished a separate program ('rebdsim') is used to calculate moments and optical functions from the distribution at each plane. This information can then be compared to an analytical description of the lattice such as that from MADX.

Note: It is important to open any apertures that are by design close to the beam such as collimators. A non-Gaussian distribution will affect the calculation of the optical parameters from the particle distribution.

6.1.1 Running BDSIM

We recommend the following settings:

- Collimators are opened to at least 6 sigma of the beam distribution at their location.
- The *stopSecondaries* and *stopTracks* options are turned on to prevents secondaries being simulated and recorded.
- The physics list is set to "" - an empty string. This leaves only magnetic field tracking so that if a particle does hit the accelerator it will pass through without scattering.
- Simulate between 1000 and 50000 particles (events).

Note: This procedure is only suited to comparing linear optical functions. If sextupoles or higher order magnets are present, these should be set to zero strength but must remain in the lattice. The pybdsim.Convert.MadxTfs2Gmad converter for example provides a boolean flag to convert the lattice with only linear optical components. The user may of course proceed with non-linear magnetic fields included but it is only useful to compare the sigma in each dimension to a similarly similuated distribution and not the Twiss parameters.

6.1.2 Analysing Optical Data

The *rebdsim* tool can be used with an input *analysisConfig.txt* that specifies *CalculateOpticalFunctions* to 1 or true in the header (see BDSIM manual). Or the specially prepared optics tool *rebdsimOptics* can be used to achieve the same outcome - we recommend this. In the terminal:

```
$> rebdsimOptics myOutputFile.root optics.root
```

This may take a few minutes to process. This analyses the file from the BDSIM run called 'myOutputFile.root' and produces another ROOT file called *optics.root* with a different structure. This output file contains only optical data.

6.2 Comparing to MADX

After preparing the optics from BDSIM, they may be compared to a MADX Tfs instance with the following command in Python (for example):

```
>>> pybdsim.Compare.MadxVsBDSIM('twiss_v5.2fs', 'optics.root')
```

This will produce a series of plots comparing the orbit, beam size, and linear optical functions.

The MADX twiss file (in tfs format) should contain all the possible columns in the Twiss Module table. This can be prepared in a similar way as we would do for converting to BDSIM GMAD syntax:

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

Note: The user should take care to ensure the emittance and energy spread (EX, EY, SIGE) are correctly specified in MADX for accurate comparison. The energy spread will contribute to the beam size in dispersive regions. The emittance will scale the beam size.

6.3 Comparing to MAD8

The comparison for MAD8 is exactly the same as MADX - please see above for further details. One difference is that both a TWISS and ENVELOPE file are required.:

```
>>> pybdsim.Compare.Mad8VsBDSIM('../mad8/TWISS_T4D', '../mad8/ENVEL_T4D', 'xfel_
↪optics.root')
```

6.4 Comparing to BDSIM

Two BDSIM optics files can also be compared with the following command:

```
>>> pybdsim.Compare.BDSIMVsBDSIM('optics1.root', 'optics2.root')
```

6.5 Comparing to PTC

BDSIM output can be compared to the PTC output from the PTC_TRACK routine available in MADX. The PTC output is written to a file typically named *trackone*, however it is necessary to convert this into a BDSIM-like ROOT output file. This can be easily accomplished with the *ptc2bdsim* tool, however the particle species and nominal momentum is required to correctly convert to the BDSIM coordinate convention. An example terminal command:

```
$> ptc2bdsim trackone ptc.root proton 0.9999
```

Once the ROOT file has been generated, the *rebdsimOptics* tool (see Analysing Optical Data) must be used to generate the ROOT file with the appropriate optical data. Finally, the two files can be compared with the following command:

```
>>> pybdsim.Compare.PTCVsBDSIM('ptc_optics.root', 'bdsim_optics.root')
```

6.6 Comparing to Transport

With the help of pytransport a TRANSPORT “FOR002” output file that has sigma matrices can be read and compared with BDSIM output:

```
>>> pybdsim.Compare.TransportVsBDSIM('FOR002.DAT', 'bdsim_optics.root')
```

DATA LOADING

Utilies to load BDSIM output data. This is intended for optical function plotting and small scale data extraction - not general analysis of BDSIM output.

7.1 Loading ROOT Data

The output optics in the ROOT file from *rebdsim* or *rebdsimOptics* may be loaded with pybdsim providing the *root_numpy* package is available.:

```
>>> d = pybdsim.Data.Load("optics.root")
```

In the case of a *rebdsim* file, an instance of the pybdsim.Data.RebdsimFile class is returned (See *RebdsimFile*). In the case of a raw BDSIM output file, an instance of the BDSIM DataLoader analysis class is returned (even in Python).

7.2 Sampler Data

Sampler data can be trivially extracted from a raw BDSIM output file

```
>>> import pybdsim
>>> d = pybdsim.Data.Load("output.root")
>>> primaries = pybdsim.Data.SamplerData(d)
```

The optional second argument to *SamplerData* can be either the index of the sampler as counting from 0 including the primaries, or the name of the sampler.

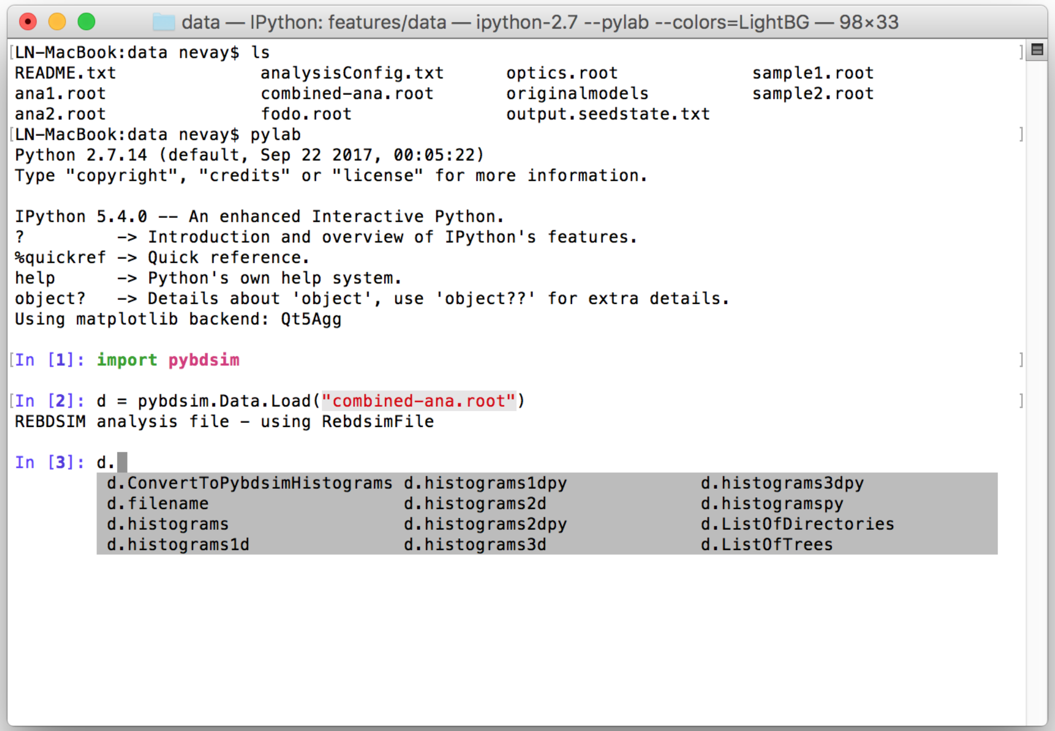
```
>>> fq15x = pybdsim.Data.SamplerData(d, fq15x)
>>> thirdAfterPrimares = pybdsim.Data.SamplerData(d, 3)
```

A near-duplicate class exists called *PhaseSpaceData* that can extract only the variables most interesting for tracking (‘x’,’xp’,’y’,’yp’,’z’,’zp’,’energy’,’t’).

```
>>> psd1 = pybdsim.Data.PhaseSpaceData(d)
>>> psd2 = pybdsim.Data.PhaseSpaceData(d, fq15x)
>>> psd3 = pybdsim.Data.PhaseSpaceData(d, 3)
```

7.3 RebdsimFile

When a *rebdsim* output file is loaded, all histograms will be loaded into a dictionary with their path inside the root file (ie in various folders) as a key. All histograms are held in a member dictionary called *histograms*. Copies are also provided in *histograms1d*, *histograms2d* and *histograms3d*.



```
LN-MacBook:data nevay$ ls
README.txt              analysisConfig.txt      optics.root             sample1.root
ana1.root               combined-ana.root       originalmodels          sample2.root
ana2.root               fodo.root               output.seedstate.txt
LN-MacBook:data nevay$ pylab
Python 2.7.14 (default, Sep 22 2017, 00:05:22)
Type "copyright", "credits" or "license" for more information.

IPython 5.4.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
Using matplotlib backend: Qt5Agg

In [1]: import pybdsim

In [2]: d = pybdsim.Data.Load("combined-ana.root")
REBDSIM analysis file - using RebdsimFile

In [3]: d.
         d.ConvertToPybdsimHistograms d.histograms1dpy             d.histograms3dpy
         d.filename                   d.histograms2d               d.histogramspy
         d.histograms                 d.histograms2dpy             d.ListOfDirectories
         d.histograms1d               d.histograms3d               d.ListOfTrees
```

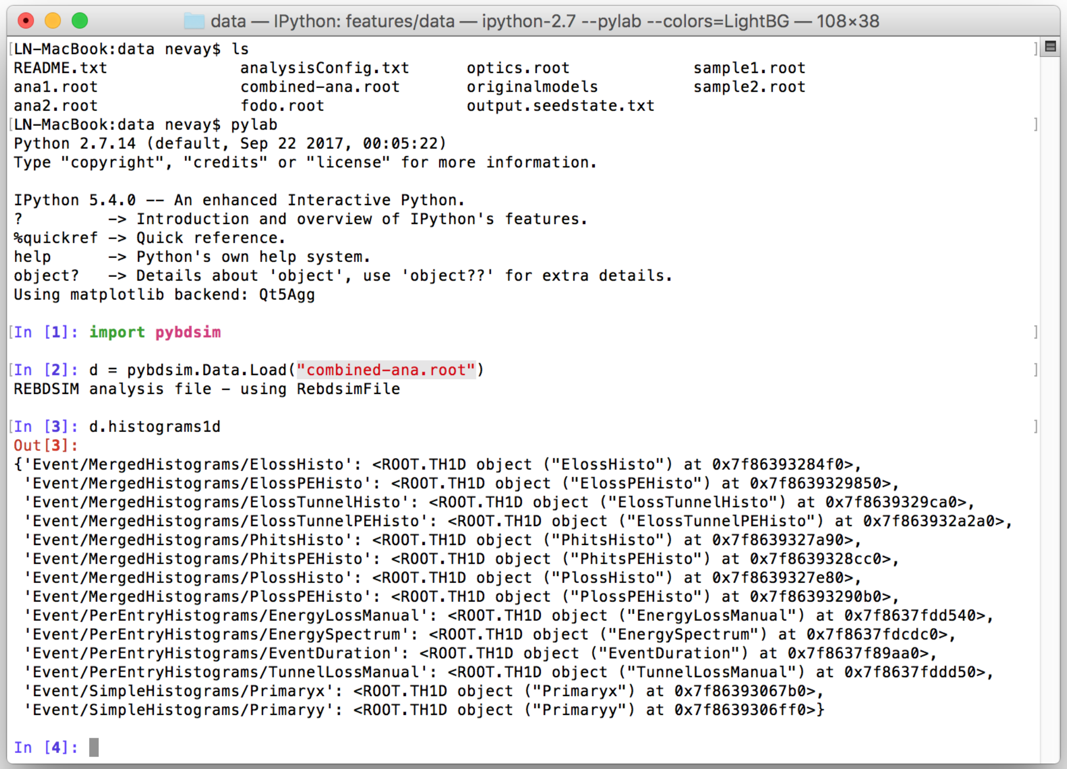
For convenience we provide wrappers for the raw ROOT histogram classes that provide easy access to the data in numpy format with simple matplotlib plotting called *pybdsim.Data.TH1*, *TH2* and *TH3*. Shown below is loading of the example output file *combined-ana.root* in *bdsim/examples/features/data*.

7.4 Histogram Plotting

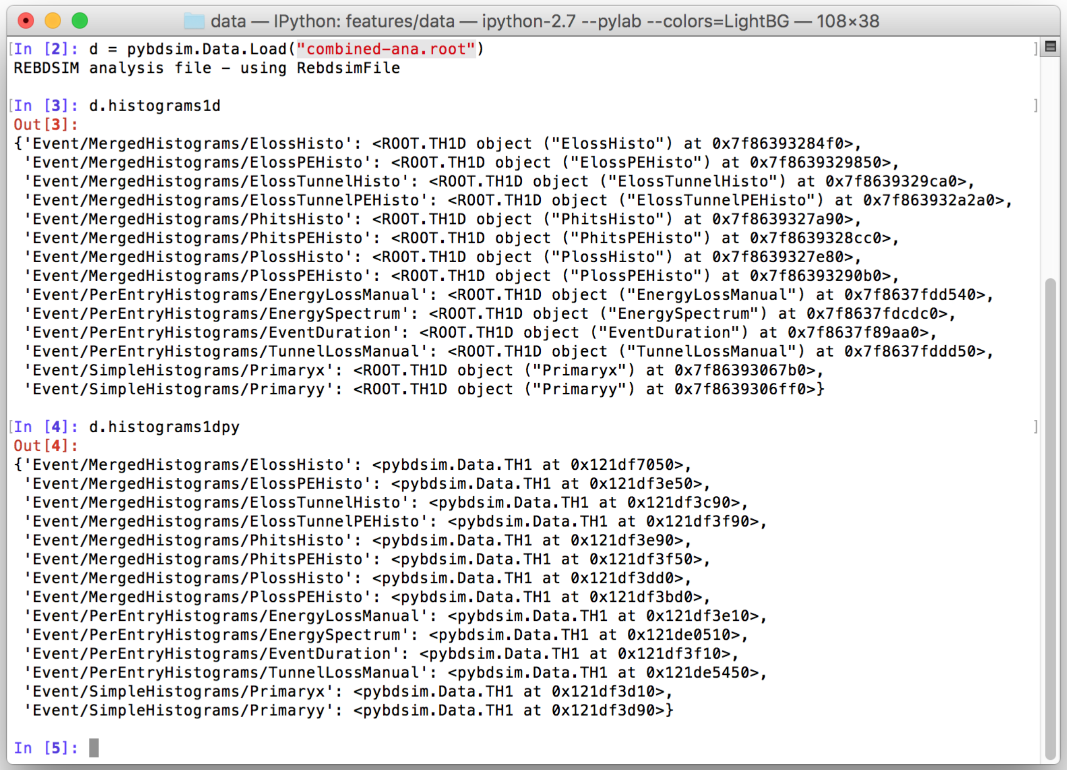
Loaded histograms that are wrapped in our pybdsim.Data.THX classes can be plotted:

```
>>> pybdsim.Plot.Histogram1D(d.histogramspy['Event/PerEntryHistograms/EnergyLossManual
↪'])
```

Note, the use of *d.histogramspy* for the wrapped set of histograms and not the raw ROOT histograms.


data — IPython: features/data — ipython-2.7 --pylab --colors=LightBG — 108×38

```
LN-MacBook:data nevay$ ls
README.txt              analysisConfig.txt      optics.root             sample1.root
ana1.root               combined-ana.root       originalmodels          sample2.root
ana2.root               fodo.root               output.seedstate.txt
LN-MacBook:data nevay$ pylab
Python 2.7.14 (default, Sep 22 2017, 00:05:22)
Type "copyright", "credits" or "license" for more information.

IPython 5.4.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
Using matplotlib backend: Qt5Agg

In [1]: import pybdsim

In [2]: d = pybdsim.Data.Load("combined-ana.root")
REBDSIM analysis file - using RebdsimFile

In [3]: d.histograms1d
Out[3]:
{'Event/MergedHistograms/ElossHisto': <ROOT.TH1D object ("ElossHisto") at 0x7f86393284f0>,
 'Event/MergedHistograms/ElossPEHisto': <ROOT.TH1D object ("ElossPEHisto") at 0x7f8639329850>,
 'Event/MergedHistograms/ElossTunnelHisto': <ROOT.TH1D object ("ElossTunnelHisto") at 0x7f8639329ca0>,
 'Event/MergedHistograms/ElossTunnelPEHisto': <ROOT.TH1D object ("ElossTunnelPEHisto") at 0x7f863932a2a0>,
 'Event/MergedHistograms/PhitsHisto': <ROOT.TH1D object ("PhitsHisto") at 0x7f8639327a90>,
 'Event/MergedHistograms/PhitsPEHisto': <ROOT.TH1D object ("PhitsPEHisto") at 0x7f8639328cc0>,
 'Event/MergedHistograms/PlossHisto': <ROOT.TH1D object ("PlossHisto") at 0x7f8639327e80>,
 'Event/MergedHistograms/PlossPEHisto': <ROOT.TH1D object ("PlossPEHisto") at 0x7f86393290b0>,
 'Event/PerEntryHistograms/EnergyLossManual': <ROOT.TH1D object ("EnergyLossManual") at 0x7f8637fdd540>,
 'Event/PerEntryHistograms/EnergySpectrum': <ROOT.TH1D object ("EnergySpectrum") at 0x7f8637fdcdc0>,
 'Event/PerEntryHistograms/EventDuration': <ROOT.TH1D object ("EventDuration") at 0x7f8637f89aa0>,
 'Event/PerEntryHistograms/TunnelLossManual': <ROOT.TH1D object ("TunnelLossManual") at 0x7f8637fddd50>,
 'Event/SimpleHistograms/Primaryx': <ROOT.TH1D object ("Primaryx") at 0x7f86393067b0>,
 'Event/SimpleHistograms/Primaryy': <ROOT.TH1D object ("Primaryy") at 0x7f8639306ff0>}

In [4]: 
```


data — IPython: features/data — ipython-2.7 --pylab --colors=LightBG — 108×38

```
In [2]: d = pybdsim.Data.Load("combined-ana.root")
REBDSIM analysis file - using RebdsimFile

In [3]: d.histograms1d
Out[3]:
{'Event/MergedHistograms/ElossHisto': <ROOT.TH1D object ("ElossHisto") at 0x7f86393284f0>,
 'Event/MergedHistograms/ElossPEHisto': <ROOT.TH1D object ("ElossPEHisto") at 0x7f8639329850>,
 'Event/MergedHistograms/ElossTunnelHisto': <ROOT.TH1D object ("ElossTunnelHisto") at 0x7f8639329ca0>,
 'Event/MergedHistograms/ElossTunnelPEHisto': <ROOT.TH1D object ("ElossTunnelPEHisto") at 0x7f863932a2a0>,
 'Event/MergedHistograms/PhitsHisto': <ROOT.TH1D object ("PhitsHisto") at 0x7f8639327a90>,
 'Event/MergedHistograms/PhitsPEHisto': <ROOT.TH1D object ("PhitsPEHisto") at 0x7f8639328cc0>,
 'Event/MergedHistograms/PlossHisto': <ROOT.TH1D object ("PlossHisto") at 0x7f8639327e80>,
 'Event/MergedHistograms/PlossPEHisto': <ROOT.TH1D object ("PlossPEHisto") at 0x7f86393290b0>,
 'Event/PerEntryHistograms/EnergyLossManual': <ROOT.TH1D object ("EnergyLossManual") at 0x7f8637fdd540>,
 'Event/PerEntryHistograms/EnergySpectrum': <ROOT.TH1D object ("EnergySpectrum") at 0x7f8637fdcdc0>,
 'Event/PerEntryHistograms/EventDuration': <ROOT.TH1D object ("EventDuration") at 0x7f8637f89aa0>,
 'Event/PerEntryHistograms/TunnelLossManual': <ROOT.TH1D object ("TunnelLossManual") at 0x7f8637fddd50>,
 'Event/SimpleHistograms/Primaryx': <ROOT.TH1D object ("Primaryx") at 0x7f86393067b0>,
 'Event/SimpleHistograms/Primaryy': <ROOT.TH1D object ("Primaryy") at 0x7f8639306ff0>}

In [4]: d.histograms1dpy
Out[4]:
{'Event/MergedHistograms/ElossHisto': <pybdsim.Data.TH1 at 0x121df7050>,
 'Event/MergedHistograms/ElossPEHisto': <pybdsim.Data.TH1 at 0x121df3e50>,
 'Event/MergedHistograms/ElossTunnelHisto': <pybdsim.Data.TH1 at 0x121df3c90>,
 'Event/MergedHistograms/ElossTunnelPEHisto': <pybdsim.Data.TH1 at 0x121df3f90>,
 'Event/MergedHistograms/PhitsHisto': <pybdsim.Data.TH1 at 0x121df3e90>,
 'Event/MergedHistograms/PhitsPEHisto': <pybdsim.Data.TH1 at 0x121df3f50>,
 'Event/MergedHistograms/PlossHisto': <pybdsim.Data.TH1 at 0x121df3dd0>,
 'Event/MergedHistograms/PlossPEHisto': <pybdsim.Data.TH1 at 0x121df3bd0>,
 'Event/PerEntryHistograms/EnergyLossManual': <pybdsim.Data.TH1 at 0x121df3e10>,
 'Event/PerEntryHistograms/EnergySpectrum': <pybdsim.Data.TH1 at 0x121de0510>,
 'Event/PerEntryHistograms/EventDuration': <pybdsim.Data.TH1 at 0x121df3f10>,
 'Event/PerEntryHistograms/TunnelLossManual': <pybdsim.Data.TH1 at 0x121de5450>,
 'Event/SimpleHistograms/Primaryx': <pybdsim.Data.TH1 at 0x121df3d10>,
 'Event/SimpleHistograms/Primaryy': <pybdsim.Data.TH1 at 0x121df3d90>}

In [5]: 
```

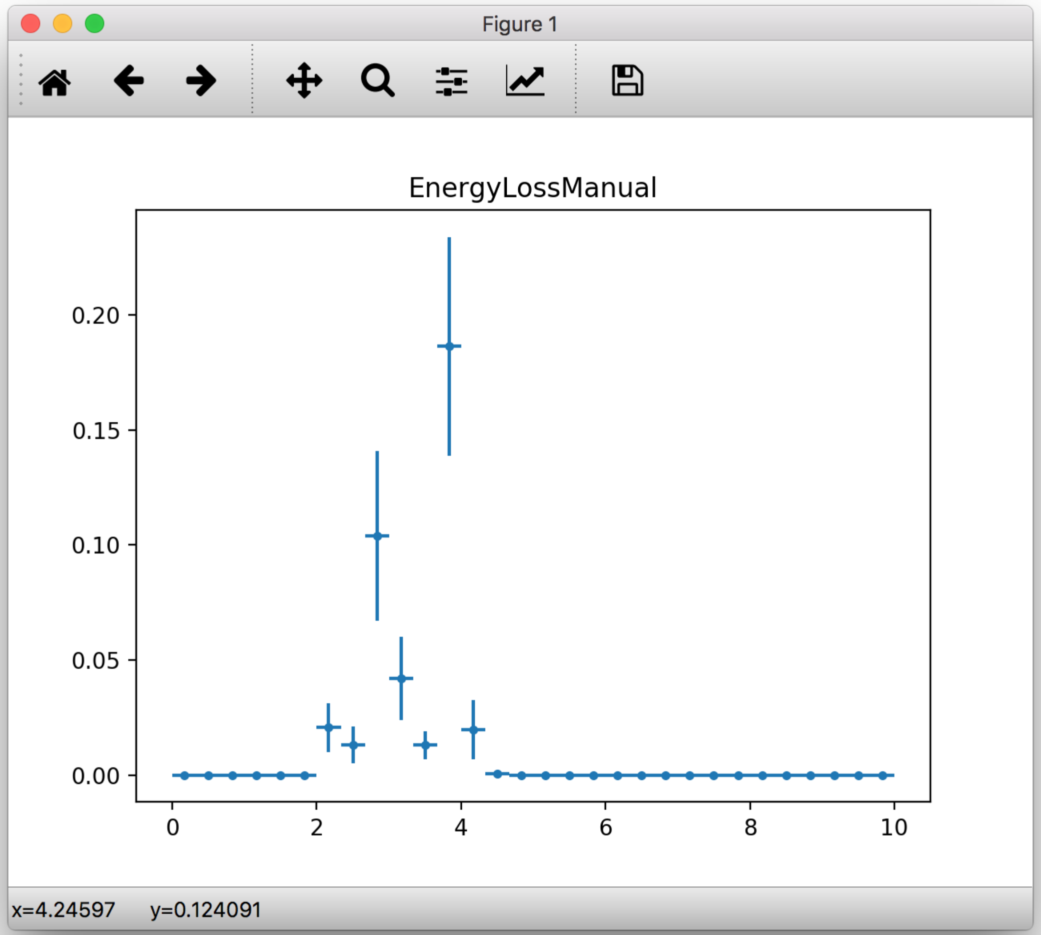


Figure 1
EnergyLossManual
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10
x=4.24597 y=0.124091

CHAPTER EIGHT

UTILITY CLASSES

Various classes are provided for the construction of BDSIM input blocks. Each class can be instantiated and then used to prepare the gmad syntax using the Python *str* or *repr* functions. These are used by the builder classes as well as the converter functions.

8.1 Beam.Beam

This beam class represents a beam definition in gmad syntax. The class has 'setter' functions that are added dynamically based on the distribution type selected.:

```
>>> b = pybdsim.Beam.Beam()
>>> b.SetParicleType("proton")
>>> b.SetDistributionType("reference")
```

8.2 Field

This module allows BDSIM format field maps to be written and loaded. There are also some plotting functions. Please see *pybdsim.Field module* for more details.

8.3 Options.Options

This class provides the set of options for BDSIM. Please see *pybdsim.Options module* for more details.

8.4 XSecBias.XSecBias

This class provides the definition process biasing in BDSIM. Please see *pybdsim.XSecBias module* for more details.

CHAPTER

NINE

SUPPORT

All support issues can be submitted to our issue tracker

9.1 Feature Request

Feature requests or proposals can be submitted to the issue tracker - select the issue type as proposal or enhancement..

Please have a look at the existing list of proposals before submitting a new one.

CHAPTER TEN

VERSION HISTORY

10.1 v2.4.0 - 2021 / 06 / 16

10.1.1 New Features

- Transform3D function in a Machine.
- Crystal, ScorerMesh and Placement also can be added to a Machine.
- Ability to insert and replace an element in a machine.

10.1.2 Bug Fixes

- Python 3.8+ warnings fixed.
- Add ROOT_INCLUDE_PATH to ROOT as newer versions don't do this automatically.
- Fixed vmin for 2D histogram plot.

10.2 v2.3.0 - 2020 / 12 / 15

10.2.1 New Features

- Convenience functions for pickling and unpickling data in the Data module with optional compression.
- Generic loss map plot.

10.3 v2.2.0 - 2020 / 06 / 08

10.3.1 New Features

- Support for Python3.

10.4 v2.1 - 2019 / 04 / 20

10.4.1 New Featuers

- Optional flag of whether to write out the converted model with *pybdsim.Convert.MadxTfs2Gmad*.
- Machine builder now supports new bdsim jcol element.
- Machine diagram drawing can now start from any arbitrary S location.
- For loaded histograms (using *pybdsim.Data.TH1*, *TH2*, *TH3* classes, there are now functions *ErrorsToSTD()* and *ErrorsToErrorOnMean()* to easily convert between the different types of error - the default is error on the mean.
- New plotting function *pybdsim.Plot.Histogram2DErrors* to visualise 2D histogram errors.

10.4.2 General

- Return arguments of *pybdsim.Convert.MadxTfs2Gmad* is now just 2 items - machine and ommitted items. Previously 3.

10.4.3 Bug Fixes

- Fix loading of Model tree from ROOT output given some recent collimation variables may have a different structure or type from the existing ones.
- In *pybdsim.Plot.Histogram2D*, the y log scale argument was "ylocscale" and is fixed to "yLogScale".

10.5 v2.0 - 2019 / 02 / 27

10.5.1 New Features

- Machine diagram plotting automatically from BDSIM output. Compatible with newer BDSIM output format.
- Support for thin R matrix, parallel transporter and thick R matrix in builder.
- Generate transfer matrix from tracking data from BDSIM for a single element.
- Control over legend location in stanard energy deposition and loss plots.
- Utility function to write sampler data from BDSIM output to a user input file.
- Support for energy variation in the beam line in MAD8 conversion.

10.5.2 General

- Remove dependency of root_numpy. pybdsim now uses only standard ROOT interfaces.
- Update physics lists.

10.5.3 Bug Fixes

- Fix bug where calling pybdsim.Plot.PrimaryPhaseSpace with an output file name would result in an error as this argument was wrongly supplied to the number of bins argument.
- Fix for hidden scientific notation when using machine diagram.
- Fix TH1 TH2 TH3 histogram x,y,z highedge variables in histogram loading. These were the lowedge duplicated, which was wrong.
- Add missing variables from sampler data given changes in BDSIM.

10.6 v1.9 - 2018 / 08 / 24

10.6.1 General

- Significant new tests.
- Trajectory loading from BDSIM ROOT output.
- Plot trajectories.
- New padding function for 1D histogram to ensure lines in plots.
- New value replacement function for histograms to ensure continuous line in log plots.
- Control over aspect ration in default 2D histogram plots.
- New classes for each element in the Builder.
- Refactor of MadxTfs2Gmad to use new classes in Builder.

10.6.2 Bug Fixes

- Fix orientation of 2D histograms in plotting.
- Fix header information labels when writing field maps with reversed order.

10.7 v1.8 - 2018 / 06 / 23

10.7.1 General

- Setup requires pytest-runner.
- Introduction of testing.
- Removed line wrapping written to GMAD files in Builder.
- “PlotBdsimOptics” is now “BDSIMOptics” in the Plot module.
- New comparison plots for arbitrary inputs from different tracking codes.
- Prepare PTC coordinates from any BDSIM sampler.

10.7.2 Bug Fixes

- Fixes for “Optics” vs “optics” naming change in ROOT files.

10.8 v1.7 - 2018 / 06 / 30

10.8.1 General

- Can specify which dimension in Field class construction (i.e. ‘x’:’z’ instead of default ‘x’:’y’).

10.8.2 Bug Fixes

- ‘nx’ and ‘ny’ were written the wrong way around from a 2D field map in pybdsim.

10.9 v1.6 - 2018 / 05 / 23

10.9.1 Bug Fixes

- Fix machine diagram plotting from BDSIM survey.
- Fix machine diagram searching with right-click in plots.

10.10 v1.5 - 2018 / 05 / 17

10.10.1 New Features

- Function now public to create beam from Madx TFS file.
- Improved searching for BDSAsciiData class.
- Ability to easily write out beam class.
- Plot phase space from any sampler in a BDSIM output file.
- __version__ information in package.
- Get a column from data irrespective of case.
- Coded energy deposition plot. Use for example for labelling cyrogenic, warm and collimator losses.
- Improved Transport BDSIM comparison.
- Function to convert a line from a TFS file into a beam definition file.

10.10.2 Bug Fixes

- Fix library loading given changes to capitalisation in BDSIM.
- Beam class now supports all BDSIM beam definitions.
- Support all aperture shapes in Builder.
- Fixes for loading optics given changes to capitalisation and BDSAsciiData class usage.
- Fixes for collimator conversion from MADX.

10.11 v1.4 - 2018 / 10 / 04

10.11.1 New Features

- Full support for loading BDSIM output formats through ROOT.
- Extraction of data from ROOT histograms to numpy arrays.
- Simple histogram plotting from ROOT files.
- Loading of sampler data and simple extraction of phase space data.
- Line wrapping for elements with very long definitions.
- Comparison plots standardised.
- New BDSIM BDSIM comparison.
- New BDSIM Mad8 comparison.
- Support for changes to BDSIM data format variable renaming in V1.0

10.11.2 Bug Fixes

- Correct conversion of all dispersion component for Beam.
- Don't write all multipole components if not needed.
- Fixed histogram plotting.
- Fixed conversion of coordinates in BDSIM2PtcInrays for subrelativistic particles.
- Fixed behaviour of fringe field *fint* and *fintx* behaviour from MADX.
- Fixed pole face angles given MADX writes out wrong angles.
- Fixed conversion of multipoles and other components for 'linear' flag in MadxTfs2Gmad.
- Fixed axis labels in field map plotting utilities.
- MADX BDSIM testing suite now works with subrelativistic particles.
- Many small fixes to conversion.

10.12 v1.3 - 2017 / 12 / 05

10.12.1 New Features

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

CHAPTER

ELEVEN

MODULE CONTENTS

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

pybdsim - python tool for BDSIM.

Module	Description
Builder	Create generic accelerators for bdsim.
Convert	Convert other formats into gmad.
Data	Read the bdsim output formats.
Fields	Write BDSIM field format.
Gmad	Create bdsim input files - lattices & options.
ModelProcessing	Tools to process existing BDSIM models and generate other versions of them.
Options	Methods to generate bdsim options.
Plot	Some nice plots for data.
Run	Run BDSIM programatically.
Visualisation	Help locate objects in the BDSIM visualisation, requires a BDSIM survey file.

Class	Description
Beam	A beam options dictionary with methods.
ExecOptions	All the executable options for BDSIM for a particular run, included in the Run module.
Study	A holder for the output of runs. Included in the Run Module.
XSecBias	A cross-section biasing object.

11.1 pybdsim.Beam module

Module containing a similarly named Beam class for creating a BDSIM beam distribution programmatically.

class pybdsim.Beam.**Beam**(*particletype='e-'*, *energy=1.0*, *distrtype='reference'*, **args*, ***kwargs*)
Bases: `dict`

ReturnBeamString()

SetDistributionType(*distrtype='reference'*)

SetE0(*e0=1*, *unitsstring='GeV'*)

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

SetParticleType(*particletype='e-'*)

SetS0(*s0=0*, *unitsstring='m'*)

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

SetX0(*x0=0.0*, *unitsstring='m'*)

SetXP0(*xp0=0.0*)

SetY0(*y0=0.0*, *unitsstring='m'*)

SetYP0(*yp0=0.0*)

SetZ0(*z0=0.0*, *unitsstring='m'*)

SetZP0(*zp0=0.0*)

WriteToFile(*filename*)

11.2 pybdsim.Builder module

class pybdsim.Builder.**Machine**(*verbose=False*, *sr=False*, *energy0=0.0*, *charge=- 1.0*)

Bases: `object`

A class represents an accelerator lattice as a sequence of components. Member functions allow various lattice components to be append to the sequence of the machine. This class allows the user to programatically create a lattice and write the BDSIM gmad representation of it.

Example:

```
>>> a = Machine()
>>> a.AddDrift('mydrift', l=1.3)
>>> a.Write("lattice.gmad")
```

Example with Sychrotron rescaling:

```
>>> a = Machine(sr=True, energy0=250,charge=-1)
>>> a.AddDipole('sb1','sbend',length=1.0,1e-5)
>>> a.AddDrift('dr1',length=1)
>>> a.AddDipole('sb2','sbend',length=1.0,1e-5)
>>> a.AddDrift("dr2",length=1)
```

Caution: adding an element of the same name twice will result the element being added only to the sequence again and not being redefined - irrespective of if the parameters are different. If verbose is used (True), then a warning will be issued.

AddBeam(*beam=None*)

Assign a beam instance to this machine. If no Beam instance is provided, a reference distribution is used.

AddBias(*biases*)

Add a XSecBias.XSecBias instance or iterable of instances to this machine.

AddCrystal(*name*, ***kwargs*)

AddCrystalCol(*name='cc'*, *length=0.01*, *xsize=0.001*, ***kwargs*)

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

AddDegrader(*length=0.1*, *name='deg'*, *nWedges=1*, *wedgeLength=0.1*, *degHeight=0.1*, *materialThickness=None*, *degraderOffset=None*, ***kwargs*)

AddDipole(*category='sbend'*)

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

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

Add a drift to the beam line

AddDump(*name='du'*, *length=0.1*, ***kwargs*)

AddECol(*name='ec'*, *length=0.1*, *xsize=0.1*, *ysize=0.1*, ***kwargs*)

AddElement(*name='el'*, *length=0.1*, *outerDiameter=1*, *geometryFile='geometry.gdml'*, ***kwargs*)

AddFodoCell(*basename*, *magnetlength*, *driftlength*, *kabs*, ***kwargs*)
basename - the basename for the fodo cell beam line elements magnetlength - length of magnets in metres driftlength - length of drift segment in metres kabs - the absolute value of the quadrupole strength - alternates between magnets

kwargs are other parameters for bdsim - ie material='Fe'

AddFodoCellMultiple(*basename='fodo'*, *magnetlength=1.0*, *driftlength=4.0*, *kabs=0.2*, *ncells=2*, ***kwargs*)

AddFodoCellSplitDrift(*basename*, *magnetlength*, *driftlength*, *kabs*, *nsplits*, ***kwargs*)
basename - the basename for the fodo cell beam line elements magnetlength - length of magnets in metres driftlength - length of drift segment in metres kabs - the absolute value of the quadrupole strength - alternates between magnets nsplits - number of segments drift length is split into

Will add qf quadrupole of strength +kabs, then drift of l=driftlength split into nsplit segments followed by a qd quadrupole of strength -kabs and the same pattern of drift segments.

nsplits will be cast to an even integer for symmetry purposes.

kwargs are other parameters for bdsim - ie aper=0.2

AddFodoCellSplitDriftMultiple(*basename='fodo'*, *magnetlength=1.0*, *driftlength=4.0*, *kabs=0.2*, *nsplits=10*, *ncells=2*, ***kwargs*)

AddGap(*name='gp'*, *length=1.0*, ***kwargs*)

AddHKicker(*name='hk'*, *hkick=0.0*, ***kwargs*)

AddIncludePost(*include*)
Add the name of a file (str) that should be included in the main file after others.

AddIncludePre(*include*)
Add the name of a file (str) that should be included in the main file before others.

AddJCol(*name='jc'*, *length=0.1*, *xsize=0.1*, *ysize=0.1*, ***kwargs*)

AddKicker(*name='kk'*, *hkick=0.0*, *vkick=0.0*, ***kwargs*)

AddLaser(*length=0.1*, *name='lsr'*, *x=1*, *y=0*, *z=0*, *waveLength=5.32e-07*, ***kwargs*)

AddMarker(*name='mk'*)
Add a marker to the beam line.

AddMuSpoiler(*name='mu'*, *length=0.1*, *b=0.0*, ***kwargs*)

AddMultipole(*name='mp'*, *length=0.1*, *knl=(0, 0)*, *ksl=(0, 0)*, ***kwargs*)

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

AddOptions(*options=None*)
Assign an options instance to this machine.

AddPlacement(*name*, ***kwargs*)

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

AddRCol(*name='rc'*, *length=0.1*, *xsize=0.1*, *ysize=0.1*, ***kwargs*)

AddRFCavity(*name='arreff'*, *length=0.1*, *gradient=10*, ***kwargs*)

AddRmat(*name='rmat'*, *length=0.1*, *r11=1.0*, *r12=0*, *r13=0*, *r14=0*, *r21=0*, *r22=1.0*, *r23=0*, *r24=0*, *r31=0*, *r32=0*, *r33=1.0*, *r34=0*, *r41=0*, *r42=0*, *r43=0*, *r44=1.0*, ***kwargs*)

AddSampler(*names*)

AddScorerMesh(*name*, ***kwargs*)

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

AddShield(*name='sh'*, *length=0.1*, ***kwargs*)

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

`AddTKicker`(*name='tk'*, *hkick=0.0*, *vkick=0.0*, ***kwargs*)

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

`AddThinRmat`(*name='rmatthin'*, *r11=1.0*, *r12=0*, *r13=0*, *r14=0*, *r21=0*, *r22=1.0*, *r23=0*, *r24=0*, *r31=0*, *r32=0*, *r33=1.0*, *r34=0*, *r41=0*, *r42=0*, *r43=0*, *r44=1.0*, ***kwargs*)

`AddTransform3D`(*name='t3d'*, ***kwargs*)

`AddUndulator`(*name='un'*, *length=1.0*, *b=0*, *undulatorPeriod=0.1*, ***kwargs*)

`AddVKicker`(*name='vk'*, *vkick=0.0*, ***kwargs*)

`AddWireScanner`(*name='ws'*, *length=0.1*, *wireDiameter=0.001*, *wireLength=0.1*, ***kwargs*)

`Append`(*item*)

`GetIntegratedAngle`()
Get the cumulative angle of all the bends in the machine. This is therefore the difference in angle between the entrance and exit vectors. All angles are assumed to be in the horizontal plane so this will not be correct for rotated dipoles.

`GetIntegratedLength`()
Get the integrated length of all the components.

`GetNamesOfType`(*category*)
Returns a list of names of elements that are of the specified category.

`InsertAndReplace`(*newElement*, *sLocation*)
New element will be placed at the central s location.

`RegenerateLenInt`()

`ReplaceElementCategory`(*category*, *newcategory*)
Change category of all elements of a given category. All parameters of the element being changed will be preserved, please update with the UpdateCategoryParameter function.

`ReplaceWithElement`(*name*, *newelement*)
Replace an element in the machine with a new element object (one of the individual element pybdsim.Builder classes that inherit the Element class).

`SynchrotronRadiationRescale`()
Rescale all component strengths for SR

`UpdateCategoryParameter`(*category*, *parameter*, *value*)
Update parameter for all elements of a given category.

`UpdateElement`(*name*, *parameter*, *value*)
Update a parameter for a specified element name. Modifying element length will produce a warning. If a value for that parameter already exists, the value will be overwritten.

`UpdateElements`(*names*, *parameter*, *value*, *namelocation='all'*)
Update multiple elements. Supplied names can be a sequence type object containing a list of element names or a string where all elements with names containing that string will be updated. namelocation specifies if names string can be at the 'beginning', 'end', or anywhere ('all') in an elements name.

`UpdateGlobalParameter`(*parameter*, *value*)
Update parameter for all elements of a given category.

`Write`(*filename*, *verbose=False*, *overwrite=True*)
Write the machine to a series of gmad files.

kwargs: overwrite : Do not append an integer to the basefilename if already exists, instead overwrite existing files.

`next`()

`class pybdsim.Builder.Line`(*name*, **args*)
Bases: `list`

A class that represents a `list` of `Elements`

Provides ability to print out the sequence or define all the components.

Example:

```
>>> d1 = Element("drift1", "drift", l=1.3)
>>> q1 = Element("q1", "quadrupole", l=0.4, k1=4.5)
>>> a = Line([d1,q1])
```

DefineConstituentElements()

Return a string that contains the lines required to define each element in the *Line*.

Example using predefined Elements name 'd1' and 'q1':

```
>>> l = Line([d1,q1])
>>> f = open("file.txt", "w")
>>> f.write(DefineConsituentElements())
>>> f.write(l)
>>> f.close()
```

class pybdsim.Builder.**Element**(*name*, *category*, ***kwargs*)

Bases: `pybdsim.Builder.ElementBase`

Element - an element / item in an accelerator beamline. Very similar to a python dict(ionary) and has the advantage that built in printing or string conversion provides BDSIM syntax.

Element(name,type,**kwargs)

```
>>> a = Element("d1", "drift", l=1.3)
>>> b = Element("qx1f", "quadrupole", l=(0.4,'m'), k1=0.2, aper1=(0.223,'m'))
>>> print(b)
qx1f: quadrupole, k1=0.2, l=0.4*m, aper1=0.223*m;
>>> str(c)
qx1f: quadrupole, k1=0.2, l=0.4*m, aper1=0.223*m\n;
```

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 - these should match BDSIM GMAD syntax.

The value can be either a single string or number or a python tuple where the second entry must be a string (shown in second example). Without specified units, the parser assumes S.I. units.

An element may also be multiplied or divided. This will scale the length and angle appropriately.

```
>>> c = Element('sb1', 'sbend', l=(0.4,'m'), angle=0.2)
>>> d = c/2
>>> print(d)
sb1: sbend, l=0.2*m, angle=0.1;
```

This inherits and extends ElementBase that provides the basic dictionary capabilities. It adds the requirement of type / category (because 'type' is a protected keyword in python) as well as checking for valid BDSIM types.

classmethod from_element(*element*)

split(*points*)

Split this element into len(points)+1 elements, with the correct lengths. This does not affect magnetic strengths, etc, which is left to derived classes where appropriate.

11.3 pybdsim.Compare

`pybdsim.Compare.`**`MadxVsBDSIM`**(*tfs*, *bdsim*, *survey=None*, *functions=None*, *postfunctions=None*, *figsize=(10, 5)*, *saveAll=True*, *outputFileName=None*)

Compares MadX and BDSIM optics variables. User must provide a tfsoptIn file or Tfsinstance and a BDSAscii file or instance.

Parameters	Description
tfs	Tfs file or pymadx.Data.Tfs instance.
bdsim	Optics root file (from rebdsimOptics or rebdsim).
survey	BDSIM model survey.
functions	Hook for users to add their functions that are called immediately prior to the addition of the plot. Use a lambda function to add functions with arguments. Can be a function or a list of functions.
figsize	Figure size for all figures - default is (12,5)

`pybdsim.Compare.`**`MadxVsBDSIMOrbit`**(*tfs*, *bds*, *survey=None*, *functions=None*, *postfunctions=None*, *figsize=(12, 5)*)

Plot both the BDSIM orbit and MADX orbit (mean x,y).

tfs - either file name or pymadx.Data.Tfs instance bds - filename or BDSAsciiData instance - rebdsimOrbit, rebdsimOptics output files

`pybdsim.Compare.`**`BDSIMVsBDSIM`**(*first*, *second*, *first_name=None*, *second_name=None*, *survey=None*, *saveAll=True*, *outputFileName=None*, ***kwargs*)

Display all the optical function plots for the two input optics files.

`pybdsim.Compare.`**`TransportVsBDSIM`**(*first*, *second*, *first_name=None*, *second_name=None*, *survey=None*, *saveAll=True*, *outputFileName=None*, ***kwargs*)

Display all the optical function plots for the two input optics files.

11.4 pybdsim.Constants module

`pybdsim.Constants.`**`GetPDGInd`**(*particlename*)

`pybdsim.Constants.`**`GetPDGName`**(*particleid*)

11.5 pybdsim.Convert

Module for various conversions.

`pybdsim.Convert.`**`BdsimPrimaries2Mad8`**(*inputfile*, *outfile*, *start=0*, *ninrays=- 1*)

” Takes .root file generated from a BDSIM run an an input and creates a MAD8 inrays file from the primary particle tree. inputfile - <str> root format output from BDSIM run outfile - <str> filename for the inrays file start - <int> starting primary particle index ninrays - <int> total number of inrays to generate

`pybdsim.Convert.`**`BdsimPrimaries2Madx`**(*inputfile*, *outfile*, *start=0*, *ninrays=- 1*)

” Takes .root file generated from a BDSIM run an an input and creates a MADX inrays file from the primary particle tree. inputfile - <str> root format output from BDSIM run outfile - <str> filename for the inrays file start - <int> starting primary particle index ninrays - <int> total number of inrays to generate, default is all available `pybdsim.Convert.`**`BdsimPrimaries2Ptc`**(*inputfile*, *outfile=None*, *start=0*, *ninrays=- 1*)

” Takes .root file generated from a BDSIM run an an input and creates a PTC inrays file from the primary particle tree. inputfile - <str> root format output from BDSIM run outfile - <str> filename for the inrays file start - <int> starting primary particle index ninrays - <int> total number of inrays to generate

`pybdsim.Convert.`**`Mad8MakeApertureTemplate`**(*inputFileName*, *outputFileName='apertures_template.dat'*)

`pybdsim.Convert.`**`Mad8MakeCollimatorTemplate`**(*inputFileName*, *outputFileName='collimator_template.dat'*)

Read Twiss file and generate template of collimator file inputFileName = “twiss.tape” outputFileName = “collimator.dat” collimator.dat must be edited to provide types and materials, apertures will be defined from lattice

`pybdsim.Convert.`**`Mad8MakeOptions`**(*inputTwissFile*, *inputEchoFile*)

`pybdsim.Convert.`**`Mad8Twiss2Gmad`**(*inputFileName*, *outputFileName*, *mad8FileName=''*, *mad8OutputFile=''*, *istart=0*, *iend=- 1*, *beam=['nominal']*, *gemit=(1e-08, 1e-08)*, *collimator='collimator.dat'*, *apertures='apertures.dat'*, *samplers='all'*, *options=True*, *flip=1*, *enableSextupoles=True*, *openApertures=True*, *openCollimators=True*, *enableSr=False*, *enableSrScaling=False*, *enableMuon=False*, *enableMuonBias=True*, *rmat=''*)

Convert MAD8 twiss output to a BDSIM model in GMAD syntax. inputfilename = mad8 TWISS output outputfilename = desired BDSIM .gmad output name istart,iend = integer number mad8 elements to begin and end conversion. beam = desired BDSIM beamtype (“reference”,”nominal”,”halo”) gemit = tuple of (emitx,emity) - default (1e-8,1e-8) - or filename of .txm with defined gemit and Esprd (and value, [name] declaration for each). collimator,apertures =relevant .dat files generated from mad8 model using pybdsim.Convert.Mad8MakeApertureTemplate & pybdsim.Convert.Mad8MakeCollimatorTemplate. rmat= mad8 r-matrix output.

`pybdsim.Convert.`**`Mad8Saveline2Gmad`**(*input*, *output_file_name*, *start_name=None*, *end_name=None*, *ignore_zero_length_items=True*, *samplers='all'*, *aperture_dict={}*, *collimator_dict='collimators.dat'*, *beam_pipe_radius=0.2*, *verbose=False*, *beam=True*, *optics=True*, *loss=True*)

`pybdsim.Convert.`**`MadxTfs2Gmad`**(*tfs*, *outputfilename*, *startname=None*, *stopname=None*, *stepsize=1*, *ignorezerolengthitems=True*, *samplers='all'*, *aperturedict={}*, *aperlocalpositions={}*, *collimatordict={}*, *userdict={}*, *partnamedict={}*, *verbose=False*, *beam=True*, *flipmagnets=None*, *usemadxaperture=False*, *defaultAperture='circular'*, *biases=None*, *allelementdict={}*, *optionsdict={}*, *beamparamsdict={}*, *linear=False*, *overwrite=True*, *write=True*, *allNamesUnique=False*, *namePrepend=''*)

MadxTfs2Gmad convert a madx twiss output file (.tfs) into a gmad tfs file for bdsim

Example:

```
>>> a,b = pybdsim.Convert.MadxTfs2Gmad('twiss.tfs', 'mymachine')
```

returns Machine, [omittedItems]

Returns two pybdsim.Builder.Machine instances. The first desired full conversion. The second is the raw conversion that’s not split by aperture. Thirdly, a list of the names of the omitted items is returned.

tfs	path to the input tfs file or pymadx.Data.Tfs instance
outputfilename	requested output file
startname	the name (exact string match) of the lattice element to start the machine at this can also be an integer index of the element sequence number in madx tfs. This item is included in the lattice
stopname	the name (exact string match) of the lattice element to stop the machine at this can also be an integer index of the element sequence number in madx tfs. This item is not included
stepsize	the slice step size. Default is 1, but -1 also useful for reversed line.
ignorezerolengthitems	nothing can be zero length in bdsim as real objects of course have some finite size. Markers, etc are acceptable but for large lattices this can slow things down. True allows to ignore these altogether, which doesn't affect the length of the machine.
samplers	can specify where to set samplers - options are None, 'all', or a list of names of elements (normal python list of strings). Note default 'all' will generate separate outputfilename_samplers.gmad with all the samplers which will be included in the main .gmad file - you can comment out the include to therefore exclude all samplers and retain the samplers file.
aperturedict	Aperture information. Can either be a dictionary of dictionaries with the the first key the exact name of the element and the daughter dictionary containing the relevant bdsim parameters as keys (must be valid bdsim syntax). Alternatively, this can be a pymadx.Aperture instance that will be queried.
aperlocalpositions	Dictionary of element indices to local aperture definitions of the form {1: [(0.0, {"APERTYPE": "CIRCULAR", "APER1": 0.4}), (0.5, {"APERTYPE": "ELLIPSE", "APER1": 0.3, "APER2": 0.4}), …], 2: […], } This defines apertures in the element at index 1 starting with a CIRCULAR aper from 0.0m (i.e. the start) before changing to ELLIPSE 0.5m into the element, with possible further changes not displayed above. As the aperture definition in GMAD is tied inseparable from its aperture definition, and vice versa, this conversion function will automatically split the element at the provided local aperture points whilst retaining optical correctness. This kwarg is mutually exclusive with "aperturedict".
collimatordict	A dictionary of dictionaries with collimator information keys should be exact string match of element name in tfs file value should be dictionary with the following keys: "bdsim_material" - the material "angle" - rotation angle of collimator in radians "xsize" - x full width in metres "ysize" - y full width in metres
userdict	A python dictionary the user can supply with any additional information for that particular element. The dictionary should have keys matching the exact element name in the Tfs file and contain a dictionary itself with key, value pairs of parameters and values to be added to that particular element.

`pybdsim.Convert.`**`MadxTfs2GmadStrength`**(*input*, *outputfilename*, *existingmachine=None*, *verbose=False*, *flipmagnets=False*, *linear=False*, *allNamesUnique=False*, *ignoreZeroLengthItems=True*)

Use a MADX Tfs file containing full twiss information to generate a strength (only) BDSIM GMAD file to be used with an existing lattice.

existingmachine	either a list or dictionary with names of elements to prepare.
flipmagnet	similar behaviour to MAdxTfs2Gmad whether to flip k values for negatively charged particles.
linear	only use linear strengths, k2 and higher set to 0.

`pybdsim.Convert._MadxTfs2Gmad.`**`MadxTfs2Gmad`**(*tfs*, *outputfilename*, *startname=None*, *stopname=None*, *stepsize=1*, *ignorezerolengthitems=True*, *samplers='all'*, *aperturedict={}*, *aperlocalpositions={}*, *collimatordict={}*, *userdict={}*, *partnamedict={}*, *verbose=False*, *beam=True*, *flipmagnets=None*, *usemadxaperture=False*, *defaultAperture='circular'*, *biases=None*, *allelementdict={}*, *optionsdict={}*, *beamparamsdict={}*, *linear=False*, *overwrite=True*, *write=True*, *allNamesUnique=False*, *namePrepend=''*)

MadxTfs2Gmad convert a madx twiss output file (.tfs) into a gmad tfs file for bdsim

Example:

```
>>> a,b = pybdsim.Convert.MadxTfs2Gmad('twiss.tfs', 'mymachine')
```

returns Machine, [omittedItems]

Returns two pybdsim.Builder.Machine instances. The first desired full conversion. The second is the raw conversion that's not split by aperture. Thirdly, a list of the names of the omitted items is returned.

tfs	path to the input tfs file or pymadx.Data.Tfs instance
outputfilename	requested output file
startname	the name (exact string match) of the lattice element to start the machine at this can also be an integer index of the element sequence number in madx tfs. This item is included in the lattice
stopname	the name (exact string match) of the lattice element to stop the machine at this can also be an integer index of the element sequence number in madx tfs. This item is not included
stepsize	the slice step size. Default is 1, but -1 also useful for reversed line.
ignorezerolengthitems	nothing can be zero length in bdsim as real objects of course have some finite size. Markers, etc are acceptable but for large lattices this can slow things down. True allows to ignore these altogether, which doesn't affect the length of the machine.
samplers	can specify where to set samplers - options are None, 'all', or a list of names of elements (normal python list of strings). Note default 'all' will generate separate outputfilename_samplers.gmad with all the samplers which will be included in the main .gmad file - you can comment out the include to therefore exclude all samplers and retain the samplers file.
aperturedict	Aperture information. Can either be a dictionary of dictionaries with the the first key the exact name of the element and the daughter dictionary containing the relevant bdsim parameters as keys (must be valid bdsim syntax). Alternatively, this can be a pymadx.Aperture instance that will be queried.
aperlocalpositions	Dictionary of element indices to local aperture definitions of the form {1: [(0.0, {"APERTYPE": "CIRCULAR", "APER1": 0.4}), (0.5, {"APERTYPE": "ELLIPSE", "APER1": 0.3, "APER2": 0.4}), …], 2: […], } This defines apertures in the element at index 1 starting with a CIRCULAR aper from 0.0m (i.e. the start) before changing to ELLIPSE 0.5m into the element, with possible further changes not displayed above. As the aperture definition in GMAD is tied inseparable from its aperture definition, and vice versa, this conversion function will automatically split the element at the provided local aperture points whilst retaining optical correctness. This kwarg is mutually exclusive with "aperturedict".
collimatordict	A dictionary of dictionaries with collimator information keys should be exact string match of element name in tfs file value should be dictionary with the following keys: "bdsim_material" - the material "angle" - rotation angle of collimator in radians "xsize" - x full width in metres "ysize" - y full width in metres
userdict	A python dictionary the user can supply with any additional information for that particular element. The dictionary should have keys matching the exact element name in the Tfs file and contain a dictionary itself with key, value pairs of parameters and values to be added to that particular element.

`pybdsim.Convert._MadxTfs2Gmad.`**`MadxTfs2GmadBeam`**(*tfs*, *startname=None*, *verbose=False*)

Takes a pymadx.Data.Tfs instance and extracts information from first line to create a BDSIM beam definition in a pybdsim.Beam object. Note that if kwarg startname is used, the optics are retrieved at the start of the element, i.e. you do not need to get the optics of the previous element, this function does that automatically.

Works for e+, e- and proton. Default emittance is 1e-9mrad if 1 in tfs file.

`pybdsim.Convert._MadxTfs2Gmad.`**`ZeroMissingRequiredColumns`**(*tfsinstance*)

Sets any missing required columns to zero. Warns user when doing so.

11.6 pybdsim.Data module

Output loading

Read bdsim output

Classes: Data - read various output files

class `pybdsim.Data.`**`ApertureInfo`**(*apertureType*, *aper1*, *aper2=0*, *aper3=0*, *aper4=0*, *offsetX=0*, *offsetY=0*)

Bases: `object`

Simple class to hold aperture parameters and extents.

class `pybdsim.Data.`**`BDSAsciiData`**(**args*, ***kwargs*)

Bases: `list`

General class representing simple 2 column data.

Inherits python list. It's a list of tuples with extra columns of 'name' and 'units'.

`ConcatenateMachine`(**args*)

Add 1 or more data instances to this one - suitable only for things that could be loaded by this class. Argument can be one or iterable. Either of str type or this class.

`Filter`(*booleanarray*)

Filter the data with a booleanarray. Where true, will return that event in the data.

Return type is BDSAsciiData

`GetColumn`(*columnstring*, *ignoreCase=False*)

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

`GetItemTuple`(*index*)

Get a specific entry in the data as a tuple of values rather than a dictionary.

`IndexFromNearestS`(*S*)

return the index of the beamline element clostest to S

Only works if "SStart" column exists in data

`MatchValue`(*parametername*, *matchvalue*, *tolerance*)

This is used to filter the instance of the class based on matching a parameter withing a certain tolerance.

```
>>> a = pybdsim.Data.Load("myfile.txt")
>>> a.MatchValue("S",0.3,0.0004)
```

this will match the "S" variable in instance "a" to the value of 0.3 within +- 0.0004.

You can therefore used to match any parameter.

Return type is BDSAsciiData

`NameFromNearestS`(*S*)

class `pybdsim.Data.`**`BeamData`**(*data*)

Bases: `object`

classmethod FromROOTFile(*path*)

class pybdsim.Data.**EventInfoData**(*data*)
Bases: `object`

Extract data from the Info branch of the Event tree.

classmethod FromROOTFile(*path*)

class pybdsim.Data.**EventSummaryData**(*data*)
Bases: *pybdsim.Data.EventInfoData*

Extract data from the Summary branch of the Event tree.

pybdsim.Data.**GetApertureExtent**(*apertureType*, *aper1=0*, *aper2=0*, *aper3=0*, *aper4=0*)

pybdsim.Data.**GetModelForPlotting**(*rootFile*, *beamlineIndex=0*)
Returns BDSAsciiData object with just the columns from the model for plotting.

pybdsim.Data.**Load**(*filepath*)
Load the data with the appropriate loader.

ASCII file - returns BDSAsciiData instance. BDSIM file - uses ROOT, returns BDSIM DataLoader instance. REBDSIM file - uses ROOT, returns RebdsimFile instance.

pybdsim.Data.**LoadPickledObject**(*filename*)
Unpickle an object. If the name contains .pbz2 the bz2 library will be used as well to load the compressed pickled object.

pybdsim.Data.**LoadROOTLibraries**()
Load root libraries. Only works once to prevent errors.

class pybdsim.Data.**ModelData**(*data*)
Bases: `object`

classmethod FromROOTFile(*path*)

GetApertureData(*removeZeroLength=False*, *removeZeroApertures=True*, *lengthTolerance=1e-06*)
return a list of aperture instances along with coordinates: l,s,x,y,apertures l - length of element s - curvilinear S coordinate at the *end* of the element x - horizontal extent y - vertical extent apertures = [ApertureInfo]

class pybdsim.Data.**OptionsData**(*data*)
Bases: `object`

classmethod FromROOTFile(*path*)

pybdsim.Data.**PadHistogram1D**(*hist*, *padValue=1e-20*)
Pad a 1D histogram with padValue.

This adds an extra 'bin' to xwidths, xcentres, xlowedge, xhighedge, contents and errors with either pad value or a linearly interpolated step in the range (i.e. for xcentres).

returns a new pybdsim.Data.TH1 instance.

class pybdsim.Data.**PhaseSpaceData**(*data*, *samplerIndexOrName=0*)
Bases: `pybdsim.Data._SamplerData`

Pull phase space data from a loaded DataLoader instance of raw data.

Extracts only: 'x','xp','y','yp','z','zp','energy','T'

Can either supply the sampler name or index as the optional second argument. The index is 0 counting including the primaries (ie +1 on the index in data.GetSamplerNames()). Examples:

```
>>> f = pybdsim.Data.Load("file.root")
>>> primaries = pybdsim.Data.PhaseSpaceData(f)
>>> samplerfd45 = pybdsim.Data.PhaseSpaceData(f, "samplerfd45")
>>> thirdAfterPrimaries = pybdsim.Data.PhaseSpaceData(f, 3)
```

`pybdsim.Data.`**`PickleObject`**`(`*ob*, *filename*, *compress=True*`)`
Write an object to a pickled file using Python pickle.

If compress is True, the bz2 package will be imported and used to compress the file.

class `pybdsim.Data.`**`ROOTHist`**`(`*hist*`)`
Bases: `object`

Base class for histogram wrappers.

`ErrorsToErrorOnMean``()`
Errors are by default the error on the mean. However, if you used ErrorsToSTD, you can convert back to error on the mean with this function, which divides by sqrt(N).

`ErrorsToSTD``()`
Errors are by default the error on the mean. Call this function to multiply by sqrt(N) to convert to the standard deviation. Will automatically only apply itself once even if repeatedly called.

class `pybdsim.Data.`**`RebdsimFile`**`(`*filename*, *convert=True*`)`
Bases: `object`

Class to represent data in rebdsim output file.

Contains histograms as root objects. Conversion function converts to pybdsim.Rebdsim.THX classes holding numpy data.

If optics data is present, this is loaded into self.Optics which is BDSAsciiData instance.

If convert=True (default), root histograms are automatically converted to classes provided here with numpy data.

`ConvertToPybdsimHistograms``()`
Convert all root histograms into numpy arrays.

`ListOfDirectories``()`
List all directories inside the root file.

`ListOfLeavesInTree``(`*tree*`)`
List all leaves in a tree.

`ListOfTrees``()`
List all trees inside the root file.

`pybdsim.Data.`**`ReplaceZeroWithMinimum`**`(`*hist*, *value=1e-20*`)`
Replace zero values with given value. Useful for log plots.

For log plots we want a small but +ve number instead of 0 so the line is continuous on the plot. This is also required for padding to work for the edge of the lines.

Works for TH1, TH2, TH3.

returns a new instance of the pybdsim.Data.TH1, TH2 or TH3.

class `pybdsim.Data.`**`SamplerData`**`(`*data*, *samplerIndexOrName=0*`)`
Bases: `pybdsim.Data._SamplerData`

Pull sampler data from a loaded DataLoader instance of raw data.

Loads all data in a given sampler.

Can either supply the sampler name or index as the optional second argument. The index is 0 counting including the primaries (ie +1 on the index in data.GetSamplerNames()). Examples:

```
>>> f = pybdsim.Data.Load("file.root")
>>> primaries = pybdsim.Data.SamplerData(f)
>>> samplerfd45 = pybdsim.Data.SamplerData(f, "samplerfd45")
>>> thirdAfterPrimaries = pybdsim.Data.SamplerData(f, 3)
```

class pybdsim.Data.**TH1**(*hist*, *extractData=True*)

Bases: *pybdsim.Data.ROOTHist*

Wrapper for a ROOT TH1 instance. Converts to numpy data.

```
>>> h = file.Get("histogramName")
>>> hpy = TH1(h)
```

class pybdsim.Data.**TH2**(*hist*, *extractData=True*)

Bases: *pybdsim.Data.TH1*

Wrapper for a ROOT TH2 instance. Converts to numpy data.

```
>>> h = file.Get("histogramName")
>>> hpy = TH2(h)
```

class pybdsim.Data.**TH3**(*hist*, *extractData=True*)

Bases: *pybdsim.Data.TH2*

Wrapper for a ROOT TH3 instance. Converts to numpy data.

```
>>> h = file.Get("histogramName")
>>> hpy = TH3(h)
```

class pybdsim.Data.**TrajectoryData**(*dataLoader*, *eventNumber=0*)

Bases: `object`

Pull trajectory data from a loaded Dataloader instance of raw data

Loads all trajectory data in a event event

```
>>> f = pybdsim.Data.Load("file.root")
>>> trajectories = pybdsim.Data.TrajectoryData(f,0)
```

next()

11.7 pybdsim.Field module

Utilities to convert and prepare field maps.

class pybdsim.Field._Field.**Field**(*array=array([], dtype=float64)*, *columns=[]*, *flip=True*, *doublePrecision=False*)

Bases: `object`

Base class used for common writing procedures for BDSIM field format.

This does not support arbitrary loop ordering - only the originally intended xyzt.

class pybdsim.Field._Field.**Field1D**(*data*, *doublePrecision=False*, *column='X'*)

Bases: *pybdsim.Field._Field.Field*

Utility class to write a 1D field map array to BDSIM field format.

The array supplied should be 2 dimensional. Dimensions are: (x,value) where value has 4 elements [x,fx,fy,fz]. So a 120 long array would have np.shape of (120,4).

This can be used for both electric and magnetic fields.

Example:

```
>>> a = Field1D(data)
>>> a.Write('outputFileName.dat')
```

class `pybdsim.Field._Field.`**`Field2D`**(*data*, *flip=True*, *doublePrecision=False*, *firstColumn='X'*, *secondColumn='Y'*)

Bases: *`pybdsim.Field._Field.Field`*

Utility class to write a 2D field map array to BDSIM field format.

The array supplied should be 3 dimensional. Dimensions are: (x,y,value) where value has 5 elements [x,y,fx,fy,fz]. So a 100x50 (x,y) grid would have np.shape of (100,50,5).

Example:

```
>>> a = Field2D(data) # data is a prepared array
>>> a.Write('outputFileName.dat')
```

The 'flip' boolean allows an array with (y,x,value) dimension order to be written as (x,y,value).

The 'doublePrecision' boolean controls whether the field and spatial values are written to 16 s.f. (True) or 8 s.f. (False - default).

class `pybdsim.Field._Field.`**`Field3D`**(*data*, *flip=True*, *doublePrecision=False*, *firstColumn='X'*, *secondColumn='Y'*, *thirdColumn='Z'*)

Bases: *`pybdsim.Field._Field.Field`*

Utility class to write a 3D field map array to BDSIM field format.

The array supplied should be 4 dimensional. Dimensions are: (x,y,z,value) where value has 6 elements [x,y,z,fx,fy,fz]. So a 100x50x30 (x,y,z) grid would have np.shape of (100,50,30,6).

Example:

```
>>> a = Field3D(data) # data is a prepared array
>>> a.Write('outputFileName.dat')
```

The 'flip' boolean allows an array with (z,y,x,value) dimension order to be written as (x,y,z,value).

The 'doublePrecision' boolean controls whether the field and spatial values are written to 16 s.f. (True) or 8 s.f. (False - default).

class `pybdsim.Field._Field.`**`Field4D`**(*data*, *flip=True*, *doublePrecision=False*)

Bases: *`pybdsim.Field._Field.Field`*

Utility class to write a 4D field map array to BDSIM field format.

The array supplied should be 5 dimensional. Dimensions are: (t,y,z,x,value) where value has 7 elements [x,y,z,t,fx,fy,fz]. So a 100x50x30x10 (x,y,z,t) grid would have np.shape of (10,30,50,100,7).

Example:

```
>>> a = Field4D(data) # data is a prepared array
>>> a.Write('outputFileName.dat')
```

The 'flip' boolean allows an array with (t,z,y,x,value) dimension order to be written as (x,y,z,t,value).

The 'doublePrecision' boolean controls whether the field and spatial values are written to 16 s.f. (True) or 8 s.f. (False - default).

11.8 pybdsim.Gmad module

Survey() - survey a gmad lattice, plot element coords
Loader() - load a gmad file using the compiled bdsim parser
GmadFile() - modify a text based gmad file

`class pybdsim.Gmad.`**`GmadFile`**`(`*fileName*`)`

Bases: `object`

Class to determine parameters and gmad include structure

`class pybdsim.Gmad.`**`GmadFileBeam`**`(`*fileName*`)`

Bases: `object`

Class to load a gmad options file to a buffer and modify the contents

`class pybdsim.Gmad.`**`GmadFileComponents`**`(`*fileName*`)`

Bases: `object`

Class to load a gmad components file to a buffer and modify the contents

Example : python> g = pybdsim.Gmad.GmadFileComponents(“./atf2_components.gmad”) python> g.change(“KEX1A”,”l”,”10”) python> g.write(“./atf2_components.gmad”)

`change``(`*element*, *parameter*, *value*`)`

Edit element dictionary

`elementNames``()`

Make a list of element names, stored in self.elementNameList

`findElement``(`*elementName*`)`

Returns the start and end (inclusive location of the element lines as a tuble (start,end)

`getParameter``(`*element*, *parameter*`)`

Edit element dictionary

`getType``(`*element*`)`

`parseElement``(`*elementString*`)`

Create element dictionary from element

`write``(`*fileName*`)`

`class pybdsim.Gmad.`**`GmadFileOptions`**`(`*fileName*`)`

Bases: `object`

Class to load a gmad options file to a buffer and modify the contents

`class pybdsim.Gmad.`**`Lattice`**`(`*filename=None*`)`

Bases: `object`

BDSIM Gmad parser lattice.

Use this class to load a bdsim input file using the BDSIM parser (GMAD) and then interrogate it. You can use this to regenerate a lattice with less information for example

```
>>> a = Lattice("filename.gmad")
```

or

```
>>> a = Lattice()
>>> a.Load("filename.gmad")
>>> a  # this will tell you some basic details
>>> print(a) # this will print out the full lattice
```

`GetAllNames``()`

GetAngle(*index*)

GetAper1(*index*)

GetAper2(*index*)

GetAper3(*index*)

GetAper4(*index*)

GetApertureExtents()

GetApertureType(*index*)

GetColumn(*column*)

GetElement(*i*)

GetIndexOfElementNamed(*elementname*)

GetKs(*index*)

GetLength(*index*)

GetName(*index*)

GetType(*index*)

IndexFromNearestS(*S*)
return the index of the beamline element clostest to S

Load(*filename*)
Load the BDSIM input file and parse it using the BDSIM parser (GMAD).

ParseLattice()
Put lattice data into python data structure

Print(*includeheaderlines=True*)

PrintZeroLength(*includeheaderlines=True*)
Print elements with zero length with s location

next()

class pybdsim.Gmad.**Survey**(*filename=None*)
Bases: `object`

Survey - load a gmad lattice and have a look

Example:

```
>>> a = Survey()
>>> a.Load('mylattice.gmad')
>>> a.Plot()
```

CompareMadX(*fileName*)

FinalDiff()

FindClosestElement(*coord*)

Load(*filename*)

Plot()

Step(*angle, length*)

11.9 pybdsim.ModelProcessing module

ModelProcessing

Tools to process existing BDSIM models and generate other versions of them.

`pybdsim.ModelProcessing.`**`GenerateFullListOfSamplers`**`(`*inputfile*, *outputfile*`)`

inputfile - path to main gmad input file

This will parse the input using the compiled BDSIM parser (GMAD), iterate over all the beamline elements and generate a sampler for every elements. Ignores samplers, but may include already defined ones in your own input.

`pybdsim.ModelProcessing.`**`WrapLatticeAboutItem`**`(`*maingmadfile*, *itemname*, *outputfilename*`)`

11.10 pybdsim.Options module

`class pybdsim.Options.`**`Editor`**`(`*fileName*`)`

Bases: `object`

`pybdsim.Options.`**`ElectronColliderOptions`**`()`

`class pybdsim.Options.`**`Options`**`(`**args*, ***kwargs*`)`

Bases: `dict`

`ReturnOptionsString``()`

`SetBLMLength``(`*length=50*, *unitsstring='cm'*`)`

`SetBLMRadius``(`*radius=5*, *unitsstring='cm'*`)`

`SetBeamPipeRadius``(`*beampiperadius=5*, *unitsstring='cm'*`)`

`SetBeamPipeThickness``(`*bpt*, *unitsstring='mm'*`)`

`SetBeamlineS``(`*beamlineS=0*, *unitsstring='m'*`)`

`SetBuildTunnel``(`*tunnel=False*`)`

`SetBuildTunnelFloor``(`*tunnelfloor=False*`)`

`SetCherenkovOn``(`*on=True*`)`

`SetChordStepMinimum``(`*csm=1*, *unitsstring='nm'*`)`

`SetDefaultBiasMaterial``(`*biases=''*`)`

`SetDefaultBiasVaccum``(`*biases=''*`)`

`SetDefaultRangeCut``(`*drc=0.7*, *unitsstring='mm'*`)`

`SetDeltaChord``(`*dc=0.001*, *unitsstring='m'*`)`

`SetDeltaIntersection``(`*di=10*, *unitsstring='nm'*`)`

`SetDeltaOneStep``(`*dos=10*, *unitsstring='nm'*`)`

`SetDontSplitSBends``(`*dontsplitsbends=False*`)`

`SetELossHistBinWidth``(`*width*`)`

`SetEMLeadParticleBiasing``(`*on=True*`)`

`SetEPAnnihilation2HadronEnhancementFactor``(`*ef=2*`)`

`SetEPAnnihilation2MuonEnhancementFactor``(`*ef=2*`)`

`SetGamma2MuonEnahncementFactor``(`*ef=2*`)`

`SetGeneralOption``(`*option*, *value*`)`

SetIncludeFringeFields(*on=True*)

SetIncludeIronMagField(*iron=True*)

SetIntegratorSet(*integratorSet='"bdsim"'*)

SetLPBFraction(*fraction=0.5*)

SetLengthSafety(*ls=10, unitsstring='um'*)

SetMagnetGeometryType(*magnetGeometryType='"none"'*)

SetMaximumEpsilonStep(*mes=1, unitsstring='m'*)

SetMaximumStepLength(*msl=20, unitsstring='m'*)

SetMaximumTrackingTime(*mtt=- 1, unitsstring='s'*)

SetMinimumEpsilonStep(*mes=10, unitsstring='nm'*)

SetNGenerate(*nparticles=10*)

SetNLinesIgnore(*nlines=0*)

SetNPerFile(*nperfile=100*)

SetOuterDiameter(*outerdiameter=2, unitsstring='m'*)

SetPhysicsList(*physicslist=''*)

SetPipeMaterial(*bpm*)

SetPrintModuloFraction(*pmf=0.01*)

SetProductionCutElectrons(*pc=100, unitsstring='keV'*)

SetProductionCutPhotons(*pc=100, unitsstring='keV'*)

SetProductionCutPositrons(*pc=100, unitsstring='keV'*)

SetRandomSeed(*rs=0*)

SetSRLowX(*lowx=True*)

SetSRMultiplicity(*srm=2.0*)

SetSamplerDiameter(*radius=10, unitsstring='m'*)

SetSensitiveBeamPipe(*on=True*)

SetSensitiveBeamlineComponents(*on=True*)

SetSenssitiveBLMs(*on=True*)

SetSoilMaterial(*sm*)

SetSoilThickness(*st=4.0, unitsstring='m'*)

SetStopSecondaries(*stop=True*)

SetStoreMuonTrajectory(*on=True*)

SetStoreNeutronTrajectory(*on=True*)

SetStoreTrajectory(*on=True*)

SetStoreTrajectoryParticle(*particle='muon'*)

SetSynchRadiationOn(*on=True*)

SetThresholdCutCharged(*tcc=100, unitsstring='MeV'*)

SetThresholdCutPhotons(*tcp=1, unitsstring='MeV'*)

SetTrackSRPhotons(*track=True*)

SetTrajectoryCutGTZ(*gtz=0.0, unitsstring='m'*)

SetTrajectoryCutLTR(*ltr=10.0, unitsstring='m'*)

SetTunnelFloorOffset(*offset=1.0, unitsstring='m'*)

SetTunnelMaterial(*tm*)

SetTunnelOffsetX(*offset=0.0, unitsstring='m'*)

SetTunnelOffsetY(*offset=0.0, unitsstring='m'*)

SetTunnelRadius(*tunnelradius=2, unitsstring='m'*)

SetTunnelThickness(*tt=1.0, unitsstring='m'*)

SetVacuumMaterial(*vm*)

SetVacuumPressure(*vp*)
Vacuum pressure in bar

SetWritePrimaries(*on=True*)

pybdsim.Options.**ProtonColliderOptions**()

11.11 pybdsim.Plot module

Useful plots for bdsim output

pybdsim.Plot.**AddMachineLatticeFromSurveyToFigure**(*figure, surveyfile, tightLayout=True, sOffset=0.0*)
Add a machine diagram to the top of the plot in a current figure

pybdsim.Plot.**AddMachineLatticeFromSurveyToFigureMultiple**(*figure, machines, tightLayout=True*)
Similar to AddMachineLatticeFromSurveyToFigure() but accepts multiple machines.

pybdsim.Plot.**AddMachineLatticeToFigure**(*figure, tfsfile, tightLayout=True*)
A forward to the pymadx.Plot.AddMachineLatticeToFigure function.

pybdsim.Plot.**Aperture**(*rootFileName, filterThin=False, surveyFileName=None*)

pybdsim.Plot.**BDSIMAperture**(*data, machineDiagram=True, plot='xy', plotApertureType=True, removeZeroLength=False, removeZeroApertures=True*)
Plot the aperture from a BDSIM DataLoader instance. By default it's colour coded and excludes any 0 aperture elements. Zero length elements are included.

pybdsim.Plot.**BDSIMApertureFromFile**(*filename, machineDiagram=True, plot='xy', plotApertureType=True, removeZeroLength=False, removeZeroApertures=True*)
Plot the aperture from a BDSIM output file. By default it's colour coded and excludes any 0s.

pybdsim.Plot.**BDSIMOptics**(*rebdsimOpticsOutput, outputfilename=None, saveall=True, survey=None, **kwargs*)
Display all the optical function plots for a rebdsim optics root file. By default, this saves all optical functions into a single (outputfilename) pdf, to save the optical functions separately, supply an outputfilename with saveall=false.

pybdsim.Plot.**DrawMachineLattice**(*axesinstance, bdsasciidataobject, sOffset=0.0*)

pybdsim.Plot.**EnergyDeposition**(*filename, outputfilename=None, tfssurvey=None, bdsimsurvey=None*)
Plot the energy deposition from a REBDSIM output file - uses premade merged histograms.

Optional either Twiss table for MADX or BDSIM Survey to add machine diagram to plot. If both are provided, the machine diagram is plotted from the MADX survey.

pybdsim.Plot.**EnergyDepositionCoded**(*filename, outputfilename=None, tfssurvey=None, bdsimsurvey=None, warmaperinfo=None, **kwargs*)
Plot the energy deposition from a REBDSIM output file - uses premade merged histograms.

Optional either Twiss table for MADX or BDSIM Survey to add machine diagram to plot. If both are provided, the machine diagram is plotted from the MADX survey.

If a BDSIM survey is provided, collimator positions and dimensions can be taken and used to split losses into categories: collimator, warm and cold based on warm aperture infomation provided. To enable this, the "warmaperinfo" option must be set according to the prescription below.

The user can supply a list of upper and lower edges of warm regions or give the path to a coulmn-formated data file with this information via the "warmaperinfo" option. Set warmaperinfo=1 to treat all non-collimator losses as warm or set warmaperinfo=-1 to treat them as cold. Default is not perform the loss classification.

If no warm aperture information is provided, the plotting falls back to the standard simple plotting provided by a pybdsimm.Plot.Hisgogram1D interface.

Args: filename (str): Path to the REBDSIM data file outputfilename (str, optional): Path where to save a pdf file with the plot. Default is None.

tfssurvey (str, optional): Path to MADX survey used to plot machine diagram on top of figure. Default is None.

tfssurvey (str, optional): Path to BDSIM survey used to classify losses into collimator/warm/cold and/or plot machine diagram on top of figure. Default is None.

warmaperinfo (int|list|str, optional): Information about warm aperture in the machine. Default is None. **kwargs: Arbitrary keyword arguments.

Kwargs: skipMachineLattice (bool): If enabled, use the BDSIM survey to classify losses, but do not plot the lattice on top.

Returns: matplotlib.pyplot.Figure object

`pybdsim.Plot.`**`Histogram1D`**(*histogram*, *xlabel=None*, *ylabel=None*, *title=None*, *scalingFactor=1.0*, *xScalingFactor=1.0*, *figsize=(10, 5)*, ***errorbarKwargs*)

Plot a pybdsim.Data.TH1 instance.

xlabel - x axis label ylabel - y axis label title - plot title scalingFactor - multiplier for values xScalingFactor - multiplier for x axis coordinates

`pybdsim.Plot.`**`Histogram1DMultiple`**(*histograms*, *labels*, *log=False*, *xlabel=None*, *ylabel=None*, *title=None*, *scalingFactors=None*, *xScalingFactors=None*, *figsize=(10, 5)*, ***errorbarKwargs*)

Plot multiple 1D histograms on the same plot. Histograms and labels should be lists of the same length with pybdsim.Data.TH1 objects and strings.

xScalingFactors may be a float, int or list

Example:

```
Histogram1DMultiple([h1,h2,h3],
                    ['Photons', 'Electrons', 'Positrons'],
                    xlabel=r'$\mu$m',
                    ylabel='Fraction',
                    scalingFactors=[1,100,100],
                    xScalingFactors=1e6,
                    log=True)
```

`pybdsim.Plot.`**`Histogram2D`**(*histogram*, *logNorm=False*, *xLogScale=False*, *yLogScale=False*, *xlabel=''*, *ylabel=''*, *zlabel=''*, *title=''*, *aspect='auto'*, *scalingFactor=1.0*, *xScalingFactor=1.0*, *yScalingFactor=1.0*, *figsize=(6, 5)*, *vmin=None*, *autovmin=False*, *vmax=None*, ***imshowKwargs*)

Plot a pybdsim.Data.TH2 instance. logNorm - logarithmic colour scale xlogscale - x axis logarithmic scale ylogscale - y axis logarithmic scale zlabel - label for color bar scale aspect - "auto", "equal", "none" - see imshow? scalingFactor - multiplier for values xScalingFactor - multiplier for x coordinates yScalingFactor - multiplier for y coordinates autovmin - fill in the background (normally white) with minimum vmin - explicitly control the vmin for the log normalisation vmax - explicitly control the vmax for the log normalisation `pybdsim.Plot.`**`Histogram2DErrors`**(*histogram*, *logNorm=False*, *xLogScale=False*, *yLogScale=False*, *xlabel=''*, *ylabel=''*, *zlabel=''*, *title=''*, *aspect='auto'*, *scalingFactor=1.0*, *xScalingFactor=1.0*, *yScalingFactor=1.0*, *figsize=(6, 5)*, ***imshowKwargs*)

`pybdsim.Plot.`**`Histogram3D`**(*th3*)
Plot a pybdsim.Data.TH3 instance - TBC

`pybdsim.Plot.`**`LossAndEnergyDeposition`**(*filename*, *outputfilename=None*, *tfssurvey=None*, *bdsimsurvey=None*, *hitslegendloc='upper left'*, *elosslegendloc='upper right'*, *perelement=False*, *elossylim=None*, *phitsylim=None*)
Load a REBDSIM output file and plot the merged histograms automatically generated by BDSIM.

Optional either Twiss table for MADX or BDSIM Survey to add machine diagram to plot.

`pybdsim.Plot.`**`LossMap`**(*ax*, *xcentres*, *y*, *ylow=None*, ***kwargs*)
Plot a loss map in such a way that works well for very large loss maps. xcentres, xwidth and y are all provided by TH1 python histograms (see pybdsim.Data.TH1).

Parameters

- **`ax`** – Matplotlib axes instance to draw to
- **`xcentres`** – centres of bins
- **`y`** – loss map signal data, same length as xcentres.
- **`ylow`** – small non-zero value to fill between to ensure works with log scales.

kwargs: * passed to calls to plot and fill_between.

`pybdsim.Plot.`**`MadxTfsBeta`**(*tfsfile*, *title=''*, *outputfilename=None*)
A forward to the pymadx.Plot.PlotTfsBeta function.

`pybdsim.Plot.`**`PhaseSpace`**(*data*, *nbins=None*, *outputfilename=None*, *extension='.pdf'*)
Make two figures for coordinates and correlations.

Number of bins chosen depending on number of samples.

'outputfilename' should be without an extension - any extension will be stripped off. Plots are saves automatically as pdf, the file extension can be changed with the 'extension' kwarg, e.g. extension='.png'.

`pybdsim.Plot.`**`PhaseSpaceFromFile`**(*filename*, *samplerIndexOrName=0*, *outputfilename=None*, *extension='.pdf'*)
Load a BDSIM output file and plot the phase space of a sampler (default the primaries). Only accepts raw BDSIM output.

'outputfilename' should be without an extension - any extension will be stripped off. Plots are saves automatically as pdf, the file extension can be changed with the 'extension' kwarg, e.g. extension='.png'.

`pybdsim.Plot.`**`PhaseSpaceSeparateAxes`**(*filename*, *samplerIndexOrName=0*, *outputfilename=None*, *extension='.pdf'*, *nbins=None*, *energy='total'*, *offsetTime=True*, *includeSecondaries=False*, *coordsTitle=None*, *correlationTitle=None*, *scalefactors={}*, *labels={}*, *log1daxes=False*, *log2daxes=False*, *includeColorbar=True*)
Plot the coordinates and correlations of both the transverse and longitudinal phase space in separate plots (four total) recorded in a sampler. Default sampler is the primary distribution.

'outputfilename' is name without extension, extension can be supplied as a string separately. Default = pdf.

The number of bins chosen depending on number of samples. Can be overridden with nbins.

Energy can be binned as either kinetic or total (default), supply either energy='total' or energy='kinetic'.

offSetTime centers the time distribution about the nominal time for the specified sampler rather than the absolute time. Default = True.

Secondaries can be included in the distributions with includeSecondaries. Default = False.

Plot titles can be supplied as strings with coordsTitle and correlationTitle.

Parameter scale factors should be supplied in a dictionary in the format {parameter: scalefactor}, e.g scalefactors={'x': 1000, 'y':1000}. Acceptable parameters are 'x','y','xp','yp', 'T','kinetic', and 'energy' for total energy.

Axis labels for parameters should be supplied as a dictionary in the format {parameter: label}, e.g labels={'x': "X (mm)", 'energy': "Energy (MeV)"}. Acceptable parameters are 'x','y','xp','yp', 'T','kinetic', and 'energy' for total energy.

log1daxes & log2daxes plots the 1D and 2D phase space on logarithmic scales respectively. Defaults = False.

includeColorbar adds a colorbar to the correlation plots. The colorbar is normalised for all plot subfigures. Default = True.

`pybdsim.Plot.`**`PlotAlpha`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotBeta`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotDisp`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotDispP`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotMean`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotNPart`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotSigma`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PlotSigmaP`**`(`*bds*, *outputfilename=None*, *survey=None*, ***kwargs*`)`

`pybdsim.Plot.`**`PrimaryPhaseSpace`**`(`*filename*, *outputfilename=None*, *extension='.pdf'*`)`

Load a BDSIM output file and plot primary phase space. Only accepts raw BDSIM output.

'outputfilename' should be without an extension - any extension will be stripped off. Plots are saves automatically as pdf, the file extension can be changed with the 'extension' kwarg, e.g. extension='.png'.

`pybdsim.Plot.`**`PrimarySurvival`**`(`*filename*, *outputfilename=None*, *tfssurvey=None*, *bdsimsurvey=None*`)`

`pybdsim.Plot.`**`PrimaryTrajectoryAndProcess`**`(`*rootData*, *eventNumber*`)`

`pybdsim.Plot.`**`ProvideWrappedS`**`(`*sArray*, *index*`)`

`pybdsim.Plot.`**`SubplotsWithDrawnMachineLattice`**`(`*survey*, *nrows=2*, *machine_plot_gap=0.01*, *gridspec_kw=None*, *subplots_kw=None*, ***fig_kw*`)`

Create a figure with a single column of axes, sharing the x-axis by default, with the machine drawn from the provided survey on the top row axes. nrows gives the number of axes, the first is always the machine lattice. by default 2 are drawn, the first for the machine, and the second for any data to be plotted to afterwards.

Parameters: survey : BSDIM survey which is used to draw the machine lattice on the top axes. machine_plot_gap : vertical space between the top of the first axes and the bottom of the machine axes. By default this is small.

Returns (figure, machine_axes, (axes1, axes2, . . .))

figure : Figure instance. machine_axes : Axes instance with the machine drawn on it. Can be used to further edit axes : iterable of axes, in order from the first below the machine, downwards.

`pybdsim.Plot.`**`Trajectory3D`**`(`*rootFileName*, *eventNumber=0*, *bottomLeft=None*, *topRight=None*`)`

Plot e-, e+ and photons only as r,g,b respectively for a given event from a BDSIM output file.

bottomLeft and topRight are optional [xlow,xhigh] limits for plots.

11.12 pybdsim.Run module

`pybdsim.Run.`**`Bdsim`**(*gmadpath*, *outfile*, *ngenerate=10000*, *batch=True*, *silent=False*, *errorSilent=False*, *options=None*, *bdsimExecutable=None*)

Runs bdsim with gmadpath as inputfile and outfile as outfile. Runs in batch mode by default, with 10,000 particles. Any extra options should be provided as a string or iterable of strings of the form "–vis_debug" or "–vis_mac=vis.mac", etc.

class `pybdsim.Run.`**`ExecOptions`**(**args*, ***kwargs*)

Bases: `dict`

`GetExecArgs`()

`GetExecFlags`()

`pybdsim.Run.`**`GetOpticsFromGMAD`**(*gmad*, *keep_optics=False*)

Get the optical functions as a BDSAsciiData instance from this GMAD file. If keep_optics is false then all intermediate files are discarded, otherwise the final optics ROOT file is written to ./

class `pybdsim.Run.`**`GmadModifier`**(*rootgmadfilename*)

Bases: `object`

`CheckExtensions`()

`DetermineIncludes`(*filename*)

`ReplaceTokens`(*tokenDict*)

`pybdsim.Run.`**`Rebdsim`**(*rootpath*, *inpath*, *outpath*, *silent=False*, *rebdsimExecutable=None*)

Run rebdsim with rootpath as analysisConfig file, inpath as bdsim file, and outpath as output analysis file.

`pybdsim.Run.`**`RebdsimHistoMerge`**(*rootpath*, *outpath*, *silent=False*, *rebdsimHistoExecutable=None*)

Run rebdsimHistoMerge

`pybdsim.Run.`**`RebdsimOptics`**(*rootpath*, *outpath*, *silent=False*)

Run rebdsimOptics

class `pybdsim.Run.`**`Study`**

Bases: `object`

A holder for multiple runs.

`GetInfo`(*index=- 1*)

Get info about a particular run.

`Run`(*inputfile='optics.gmad'*, *output='rootevent'*, *outfile='output'*, *ngenerate=1*, *bdsimcommand='bdsim-devel'*, ***kwargs*)

`RunExecOptions`(*execoptions*, *debug=False*)

11.13 pybdsim.Visualisation module

class `pybdsim.Visualisation.`**`Helper`**(*surveyFileName*)

Bases: `object`

To help locate objects in the BDSIM visualisation, requires a BDSIM survey file

`draw`()

Quick survey drawing for diagnostic reasons

`findComponentCoords`(*componentName*)

Returns the XYZ coordinates of a component relative to the centre

`getWorldCentre`(*type='linear'*)

Returns the center in world coordinates of the centre of the visualisation space

11.14 pybdsim.Writer module

Writer

Write files for a pybdsim.Builder.Machine instance. Each section of the written output (e.g. components, sequence, beam etc.) can be written in the main gmad file, written in its own separate file, or called from an external, pre-existing file.

Classes: File - A class that represents each section of the written output - contains booleans and strings. Writer - A class that writes the data to disk.

class pybdsim.Writer.**FileSection**(*willContain=''*)

Bases: object

A class that represents a section of a gmad file. The sections that this class can represent are:

- Components
- Sequence
- Samplers
- Beam
- Options
- Bias

The class contains booleans and strings relating to the location of that sections data. The section can set to be:

- Written in its own separate file (default)
- Written in the main gmad file
- Called from an external file

These classes are instantiated in the writer class for each section. An optional string passed in upon class instantiation is purely for the representation of the object which will state where the data will be written/called. This string should be one of the section names listed above.

Example:

```
>>> beam = FileSection('beam')
>>> beam.CallExternalFile('../myBeam.gmad')
>>> beam
pybdsim.Writer.File instance
File data will be called from the external file:
../myBeam.gmad
```

CallExternalFile(*filepath=''*)

WriteInMain()

WriteSeparately()

class pybdsim.Writer.**Writer**

Bases: object

A class for writing a pybdsim.Builder.Machine instance to file.

This class allows the user to write individual sections of a BDSIM input file (e.g. components, sequence, beam etc.) or write the machine as a whole.

There are 6 attributes in this class which are FileSection instances representing each section of the data. The location where these sections will be written/read is stored in these instances. See the FileSection class for further details.

The optional boolean 'singlefile' in the WriteMachine function for writing the sections to a single file overrides any sections locations set in their respective FileSection instances.

This class also has individual functions (e.g. WriteBeam) to write each file section and the main file (WriteMain) separately. These section functions must be called BEFORE the WriteMain function is called otherwise the main file will have no reference to these sections.

Examples:

Writing the Builder.Machine instance myMachine to separate files:

```
>>> a = Writer()
>>> a.WriteMachine(myMachine,'lattice.gmad')
Lattice written to:
lattice_components.gmad
lattice_sequence.gmad
lattice_beam.gmad
lattice.gmad
All included in main file:
lattice.gmad
```

Writing the Builder.Machine instance myMachine into a single file:

```
>>> a = Writer()
>>> a.WriteMachine(myMachine,'lattice.gmad',singlefile=True)
Lattice written to:
lattice.gmad
All included in main file:
lattice.gmad
```

WriteBeam(*machine*, *filename=''*)
Write a machines beam to disk: filename.gmad

Machine can be either a pybdsim.Builder.Machine instance or a pybdsim.Beam.Beam instance.

WriteBias(*machine*, *filename=''*)
Write the machines bias to disk: filename.gmad

WriteComponents(*machine*, *filename=''*)
Write the machines components to disk: filename.gmad

WriteMachine(*machine(machine)*, *filename(string)*, *singlefile(bool)*, *verbose(bool)*)
Write a machine to disk. By default, the machine will be written into the following individual files:

filename_components.gmad	component files (max 10k per file)
filename_sequence.gmad	lattice definition
filename_samplers.gmad	sampler definitions (max 10k per file)
filename_options.gmad	options
filename_beam.gmad	beam definition
filename_bias.gmad	machine biases (if defined)
filename.gmad	suitable main file with all sub files in correct order

These are prefixed with the specified filename / path

The optional bool singlefile = True will write all the above sections into a single file:

filename.gmad

kwargs: overwrite : Do not append an integer to the basefilename if already exists, instead overwrite existing files.

WriteMain(*machine(machine)*, *filename(string)*)
Write the main gmad file: filename.gmad

The functions for the other sections of the machine (components,sequence,beam,options,samplers,bias) must be written BEFORE this function is called.

WriteObjects(*machine*, *filename=''*)
Write the machines objects (e.g. crystals) to disk: filename.gmad

WriteOptions(*machine*, *filename=''*)
Write a machines options to disk: filename.gmad

Machine can be either a pybdsim.Builder.Machine instance or a pybdsim.Options.Options instance.

WriteSamplers(*machine*, *filename=''*)
Write the machines samplers to disk: filename.gmad

WriteSequence(*machine*, *filename=''*)
Write the machines sequence to disk: filename.gmad

11.15 pybdsim.XSecBias module

class pybdsim.XSecBias.**XSecBias**(*name*, *particle*, *processes*, *xsecfactors*, *flags*)
Bases: `object`

A class for containing all information regarding cross section definitions.

CheckBiasedProcesses()

SetFlags(*flags*)
Set flags. flags should be a space-delimited string of integers, 1-3, in the same order as the processes,

SetName(*name*)
Set the bias name. Cannot be any upper/lowercase variant of reserved keyword “xsecBias”.

SetParticle(*particle*)
Set the particle for bias to be associated with.

SetProcesses(*processes*)
Set the list of processes to be biased. processes hould be a space-delimited string of processes.

SetXSecFactors(*xsecs*)
Set cross section factors. xsecs should be a space-delimited string of floats, e.g. “1.0 1e13 1234.9”

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX