pybdsim Documentation

Release 1.2

Royal Holloway

Sep 14, 2017

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

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').

3.1 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
```

3.2 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.

3.3 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.

3.4 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')
```

3.5 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.

3.6 kwargs - Flexibility

'kwargs' are optional keyword arguments in Python. This allows the user to supply arbitrary options to a function that can be instpected 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

FOUR

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*
  - *Mad8Saveline2Gmad*
- Transport to BDSIM
  - *pytransport*
- BDSIM Primary Particle Conversion
  - *BDSIM Primaries To Others*

4.1 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

4.1.1 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.

4.1.2 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 typically two objects, which are the `pybdsim.Builder.Machine` instance and a list of any ommitted items by name.:

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

The user may convert only part of the input model by specifying *startname* and *stopname*. The full list of options is described in *pybdsim.Convert*.

Generally speaking, extra information can be folded into the conversion by the user supply a 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)
```

Note: The name must match the name given in the MADX file exactly.

Specific arguments may be given for aperture (*aperturedict*), or for collimation (*collimatordict*), which are used specifically for those purposes.

There are quite a few options and these are described in *pybdsim.Convert*.

Note: The BDSIM-provided pymadx package is required for this conversion to work.

Note: The converter will alter the names to remove forbidden characters in names in BDSIM such as '$' or '!'.

4.1.3 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')
```

4.2 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.

4.3 Mad8Twiss2Gmad

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

4.4 Mad8Saveline2Gmad

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

4.5 pytransport

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

4.6 BDSIM Primaries To Others

The primary particle coordinates generated by BDSIM may be read from an output ROOT file and written to another format to ensure the exact same coordinates are used in both simulations. This is typically used for comparison with PTC.

CHAPTER

FIVE

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.

5.1 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 anlytical 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.

5.1.1 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.

5.1.2 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.

5.2 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.

5.3 Comparing to MAD8

5.4 Comparing to Transport

CHAPTER SIX

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.

6.1 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")
```

CHAPTER

SEVEN

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.

7.1 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")
```

7.2 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.

7.3 Options.Options

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

7.4 XSecBias.XSecBias

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

CHAPTER

EIGHT

PARSER INTERACTION

CHAPTER

NINE

PLOTTING

CHAPTER

TEN

RUNNING BDSIM FROM PYTHON

CHAPTER

ELEVEN

VISUALISATION

CHAPTER

TWELVE

SUPPORT

All support issues can be submitted to our issue tracker

12.1 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

THIRTEEN

MODULE CONTENTS

This documentation is automatically generated by scanning all the source code. Parts may be incomplete. pybdsim - python tool for BDSIM

Dependency	Minimum Version Required
numpy	1.7.1
matplotlib	1.3.0
pymadx	latest

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.

13.1 pybdsim.Beam module

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

Bases: `dict`

`ReturnBeamString`()

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

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

`SetParticleType`(*particletype='e-'*)

`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*)

13.2 pybdsim.Builder module

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

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 (*biasobject*)

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

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*)

AddHKicker (*name='hk'*, *hkick=0.0*, ***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*, *ksl=0*, *tilt=0.0*, ***kwargs*)

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

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

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*)

AddSampler (**elementnames*)

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*, *ksl=0*, *tilt=0.0*, ***kwargs*)

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

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

Append (*object*)

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.

SynchrotronRadiationRescale ()
Rescale all component strengths for SR

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

13.3 pybdsim.Compare

pybdsim.Compare.**MadxVsBDSIM**(*tfs*, *bdsim*, *survey=None*, *functions=None*, *postfunctions=None*, *figsize=(12, 5)*)
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*, *bdsim*, *survey=None*, *functions=None*, *postfunctions=None*)

pybdsim.Compare.**BDSIMVsBDSIM**(*first*, *second*, *first_name=None*, *second_name=None*, *survey=None*, ***kwargs*)
Display all the optical function plots for the two input optics files.

pybdsim.Compare.**TransportVsBDSIM**(*parameter*, *bdsfile*, *transfile*, *transscaling=1*, *lattice=None*, *ylabel=None*, *outputfilename=None*)

13.4 pybdsim.Constants module

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

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

13.5 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*, *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, istart=0, beam=['nominal'], gemit=(1e-10, 1e-10), mad8FileName='', collimator='collimator.dat', apertures='apertures.dat', samplers='all', options=True, flip=1, enableSextupoles=True, enableOctupoles=True, enableDecapoles=True, openApertures=True, openCollimators=True, enableDipoleTiltTransform=True, enableDipolePoleFaceRotation=True, enableSr=False, enableSrScaling=False, enableMuon=False, enableMuonBias=True*)

Convert MAD8 twiss output to a BDSIM model in GMAD syntax.

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**(*input*, *outputfilename*, *startname=None*, *stopname=None*, *stepsize=1*, *ignorezerolengthitems=True*, *thinmultipoles=True*, *samplers='all'*, *aperturedict={}*, *collimatordict={}*, *userdict={}*, *beampiperadius=5.0*, *verbose=False*, *beam=True*, *flipmagnets=None*, *usemadxaperture=False*, *defaultAperture='circular'*, *biases=None*, *allelementdict={}*, *optionsDict={}*, *linear=False*)

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

inputfilename	path to the input file
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.
thinmultipoles	will convert thin multipoles to ~1um thick finite length multipoles with upscaled k values - experimental feature
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.
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.
beampiperadius	In metres. Default beam pipe radius and collimator setting if unspecified.
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 - BDISM however, follows their manual 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.
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.

Example:

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

In normal mode: Returns Machine, [omittedItems]

In verbose mode: Returns Machine, 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.

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

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

`pybdsim.Convert._MadxTfs2Gmad.`**`MadxTfs2Gmad`**(*input*, *outputfilename*, *startname=None*, *stopname=None*, *stepsize=1*, *ignorezerolengthitems=True*, *thinmultipoles=True*, *samplers='all'*, *aperturedict={}*, *collimatordict={}*, *userdict={}*, *beampiperadius=5.0*, *verbose=False*, *beam=True*, *flipmagnets=None*, *usemadxaperture=False*, *defaultAperture='circular'*, *biases=None*, *allelementdict={}*, *optionsDict={}*, *linear=False*)

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

inputfilename	path to the input file
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.
thinmultipoles	will convert thin multipoles to ~1um thick finite length multipoles with upscaled k values - experimental feature
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.
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.
beampiperadius	In metres. Default beam pipe radius and collimator setting if unspecified.
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 - BDISM however, follows their manual 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.
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.

Example:

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

In normal mode: Returns Machine, [omittedItems]

In verbose mode: Returns Machine, 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.

`pybdsim.Convert._MadxTfs2Gmad.`**`ZeroMissingRequiredColumns`**`(`*tfsinstance*`)`
Sets any missing required columns to zero. Warns user when doing so.

13.6 pybdsim.Data module

Output

Read bdsim output

Classes: Data - read various output files

`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*`)`
This is used to concatenate machines.

`Filter``(`*booleanarray*`)`
Filter the data with a booleanarray. Where true, will return that event in the data.

Return type is BDSAsciiData

`GetColumn``(`*columnstring*`)`
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*`)`
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*`)`

`pybdsim.Data.`**`Load`**`(`*filepath*`)`

13.7 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*)

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*)

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*)

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).

13.8 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*)

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*)

13.9 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*)

13.10 pybdsim.Options module

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

pybdsim.Options.**ElectronColliderOptions**()

pybdsim.Options.**MinimumStandard**()

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'*)

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*)

SetStopTracks(*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**()

13.11 pybdsim.Plot module

Useful plots for bdsim output

pybdsim.Plot.**AddMachineLatticeFromSurveyToFigure**(*figure, *args, **kwargs*)

kwargs - 'tightLayout' is set to True by default - can be supplied in kwargs to force it to false.

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

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

pybdsim.Plot.**MadxTfsBetaSimple**(*tfsfile, title='', outputfilename=None*)
A forward to the pymadx.Plot.PlotTfsBetaSimple function.

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

13.12 pybdsim.Run module

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

GetExecArgs()

GetExecFlags()

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

CheckExtensions()

DetermineIncludes (*filename*)

ReplaceTokens (*tokenDict*)

class pybdsim.Run.**Study**

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*)

13.13 pybdsim.Visualisation module

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

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

13.14 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="*)

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**

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

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.

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

13.15 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

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX