VisBOUTIt Users Manual

P. Naylor , University of York

August 2015

Contents

1 Introduction

This document outlines how to use the Python visualisation tool, VisBOUTIt. This tool imports and displays a variable from BOUT++ data in a specified geometry and can render an image sequence with the visualisation package VisIt. VisIt is a visualisation package available from the Lawrence Livermore National Laboratory (LLNL), the site also contains documentation on how to use the VisIt software. This package can import the following from BOUT++ data; specified scalar variables, field parallel vectors and values from the eigen solver. The VisBOUTIt package can convert to the following coordinate systems; Cylindrical geometry Figure (1), Field aligned Toroidal Geometry with Poloidal Planes (Torus) Figure (2) and Field aligned Toroidal Geometry with Toroidal Planes (ELM) Figure (3). This manual is divided into an overview on how the package works, instructions on how to use the package and some examples. Where appropriate some commands are used and are in the format shown below.

```
print "Hello World"
```

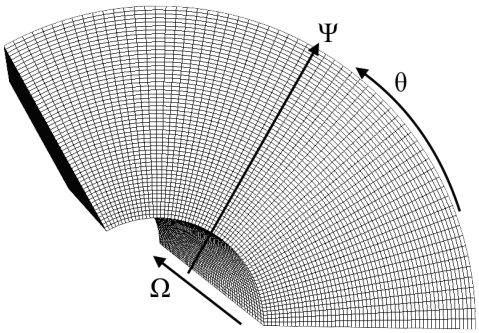

Figure 1: Cylindrical Mesh (Ψ, Ω, θ)

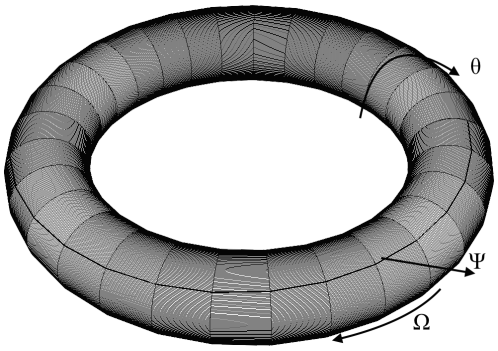

Figure 2: Torus Mesh (Ψ, Ω, θ)

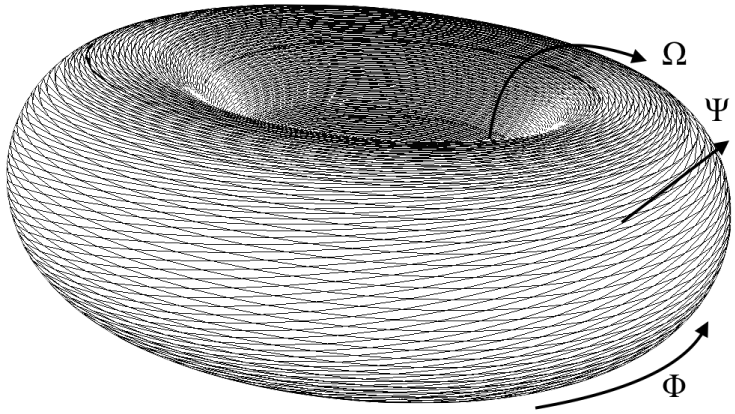

Figure 3: ELM Mesh (Ψ, Ω, Φ)

1.1 Coordinate Systems

The Cylindrical coordinate system (Ψ, Ω, θ) , detailed in Figure (1) where;

- Ψ Radial coordinate
- Ω Field parallel coordinate (Height of Cylinder)
- θ Azmuthal cooridnate

The Torus coordinate system (Ψ, Ω, θ) , detailed in Figure (2) where:

- Ψ Radial coordinate
- Ω Field parallel coordinate (Toroidal coordinate)
- θ Poloidal coordinate

The ELM coordinate system (Ψ, Ω, Φ) , detailed in Figure (3) where:

- Ψ Radial coordinate
- Ω Field parallel coordinate (Poloidal coordinate)
- Φ Toroidal coordinate

2 How the package works

VisBOUTIt utilises the Python libraries from VisIt, PyEVTK and BOUT++ to import, plot and export an image sequence of BOUT++ data. This works by importing the user specified variable from the BOUT++ data, converting the coordinate system and writing this to time stamped files using the VTK format. VisBOUTIt can import the data into VisIt allowing the user to setup a desired plot before the image sequence is rendered. VisBOUTIt can be used to visualise; scalar, field parallel vector variables and values from the eigensolver. All the imported values are the raw values from the BOUT++ files, the plotted data does not contain normalisation factors. Within the seperate functions except the eigen functions, there is a capabiltity to find the maximum and minimum of the imported data. This can be used to set the scale for the entire image sequence, instead of using maximum and minimum values for each time slice generated by VisIt.

The basic procedure for the VisBOUTIt package is detailed below.

1. Run the **visitsetup** script.
2. A variable from the BOUT++ data is converted to the desired coordinate system.
3. The converted data is plotted in a VisIt session automatically or manually.
4. The dataset is rendered out as an image sequence.

2.1 Requirements

- The package VisBOUTIt should be located in the BOUT++ directory under */tools/pylib/*, if VisBOUTIt is not present then update the BOUT repository.
- PyEVTK python library, which is available from:

  https://pypi.python.org/pypi/PyEVTK or at https://bitbucket.org/pauloh/pyevtk. And can be installed using pip, using the command below.

  ```
  pip install pyevtk
  ```

- An installation of VisIt, available from the LLNL website:

  https://wci.llnl.gov/simulation/computer-codes/visit

- The user's bash profile should contain the python path to BOUT's pylib tools directory. For example the bash profile should contain something similar to the line below. Where usr is placeholder for the path to the directory that contains BOUT-dev.

  ```
  export PYTHONPATH=/usr/BOUT-dev/tools/pylib/:$PYTHONPATH
  ```

2.2 Package Contents

The package VisBOUTIt is located within the BOUT++ python tools directory, *BOUT-dev/tools/pylib/visboutit/*. And contains the python files; **circle.py, draw.py, eigen.py, image.py, vector.py, visitsetup, visual.py** and **vtk.py**.

2.2.1 circle.py

The script **circle.py** contains functions to create a Torus mesh and is used by the VTK writing functions.

2.2.2 draw.py

The **draw.py** locates the vtk files in the *batch* directory and presents the data in a VisIt window for the user to orientate the data. The python interface prompts to save a VisIt session of the current view, which is saved in the data directory. This VisIt session is used to import the user's desired plot settings for rendering out the image sequence. Next the script renders out the entire set of vtk files as an image sequence stored under the directory *images/variablename_sessionname*.

The **draw.py** library contains functions to display converted data in a VisIt session, allowing the user to customise the view before rendering an image sequence.

2.2.3 eigen.py

The script **eigen.py** contains functions used to select specific eigenvectors that are then converted to either a; Cylindrical Fig (1), Torus Fig (2) or ELM Fig (3) cooridinate system.

2.2.4 image.py

The **image.py** script opens a VisIt session and renders out an image sequence into the directory *images/variablename_sessionname*.

2.2.5 vector.py

The **vector.py** contains functions that are used to plot the parallel field vectors along the field lines of the coordinate system.

2.2.6 visisetup

The **visitsetup** script finds the users VisIt directory or prompts for the directory if it fails. Also the script asks for the desired image dimensions. The VisIt directory locations and image dimensions are stored in a *visit.ini* text file within the VisBOUTIt directory which can be modified later by the user.

2.2.7 visual.py

The **visual.py** file is a library of common functions used by the VisBOUTIt package.

2.2.8 vtk.py

The script **vtk.py** contains functions to convert BOUT++ data into; cylindrical, toroidal or ELM coordinates respectively and write the corresponding VTK files. These functions place the converted data into a *batch* subdirectory of the raw data. The process of creating *batch* and *image* directories is handled within the various functions.

3 Initial Setup

Before any data can be imported the **visitsetup** script needs to be run. This looks for the VisIt directory and writes the file locations to a text file. If the script cannot find the VisIt directories it will prompt for the file paths. The text file is written to VisBOUTIt directory and can be executed with command below (when in the VisBOUTIt directory).

```
$ ./visitsetup
```

The script will also prompt for the image dimensions for the image sequences, these variables are stored within the *visit.ini* file. This file can be modified if alternate resolutions are required.

4 Scalar Variables

VisBOUTIt can be used like the showdata and plotdata functions. This section details the procedures for the package. Examples on how to use these functions are contained within this section and in Appendix (A).

4.1 Converting Scalar data:

The data conversion is handled by three seperate python functions *cylinder* , *torus* , *elm* within the **vtk.py** file. The functions can be called within the working directory that contains the data. If the working directory does not contain the data then the path argument must be completed. The converted data is written to a batch folder within the directory of the data, for plotting later.

If the python path to *BOUT-dev/tools/pylib/* has been added to the bash_profile for the user then using the following command will import the functions.

```
$ from visboutit import vtk
```

4.1.1 ELM

The *elm* function is designed to convert data to ELM geometry Figure (3). The function has the following arguments;

```
elm(name , time , zShf_int_p = 0.25 , path = None , skip = 1)
```

- **Name:** Name of variable, (string).
- **Time:** End time slice to be converted. If the time value is greater than the maximum avaliable time value then the maximum value will be used. Also using -1 will set the time variable to the maximum time value. (integer)
- **zShf_int_p:** Zshift interpolation percentage (float, $0 \leq zShf\_int\_p \leq 1$) , zShf_int_p is 25% by default. This value is used to find the tolerance between the maximum and minimum zshift for linear interpolation in the y direction only. The larger the percentage the higher the tolerance value becomes therefore fewer interpolations are performed. The zshift tolerance is calculated by equation (1)

$$z_{tol} = min_{zshift} \ + \ ((max_{zshift} - \ min_{zshift}) \ \times \ zShf\_int\_p) \tag{1}$$

  The linear interpolation is performed when an array element set by the Pseudocode below is greater than zero. The integer determines how many points are to be inserted between the two original y slices.

  ```
  int(abs( (zshift[y] - zshift[y+1] / z_tol ))
  ```

- **Path:** The path to the BOUT++ data, this can be left blank if the the function has been called within the directory that contains the data. (String)
- **Skip:** The gap between time slices. For example a skip value of 10 will convert the t = 0 , 10 , 20, etc slices. (Integer)

Examples: The example below will import the variable *"vort"* for every 10th time value across the entire time range.

```
$ vtk.elm('vort', -1 , skip = 10)
```

The following example demonstrates how to import data not located within the working directory.

```
$ vtk.elm('vort', -1 , path = '/path/to/data/')
```

4.1.2 Torus

The *torus* function is designed to convert data to Toroidal geometry and has the arguments shown below. **name, time, path** and **skip** are the same as for the *elm* function.

```
torus(name , time , step = 0.5 , skip = 1, path = None, R = None, r = None ,
dr = None , Bt = None , q = None , isttok = False)
```

- **Step:** The split between points when the function interpolates the data in the y direction. The default value is set to 0.5, if the value is set to 0 then the function will display the raw data without interpolation. (Float $0 \leq Step \leq 1$)
- **R , r , dr , Bt , q:** Settings for generating the Torus mesh. Where R: Major radius, r: Minor radius , dr: Radial width of domain, Bt: Toroidal magnetic field and q: Safety factor. These values require floats.
- **isttok:** Boolean, if True then the ISTTOK specifications are used which are defined below in Equation (2).

$$R,\ r,\ dr,\ Bt,\ q = \ 0.46,\ 0.085,\ 0.02,\ 0.5,\ 5 \tag{2}$$

- **default:** (Boolean), if True then the default speicifications defined in Equation (3) will be used to generate the Torus mesh.

$$R,\ r,\ dr,\ Bt,\ q = \ 2.0,\ 0.2,\ 0.05,\ 1.0,\ 5.0 \tag{3}$$

Examples: An example of the *torus* function importing the variable 'P' from t = 0 to t = 50, with the ISTTOK specification is shown below.

```
$ vtk.torus('P' , 50 , isttok = True)
```

4.1.3 Cylinder

The *cylinder* function is designed to convert data to cylindrical geometry and has the arguments below. The arguments **name , time , step , path** and **skip** are the same as the *torus* function.

```
cylinder(name , time , pi_fr = (2./3.) , step = 0.5 , path = None , skip = 1)
```

- **pi_fr:** The fraction of π that the cylinder has been rotated through, i.e. 2π is a full cylinder. The default value is 2/3 . If the value given is above 2π then interger values of 2π are subtracted until the value is between zero and 2π then this value is used. (Float)

Examples: The following example demonstrates importing a set of cylindrical data for all time slices. The cylindrical mesh has a rotation of $\frac{4}{3}\pi$, where the step between y slices is set to 0.4.

```
$ vtk.cylinder('vort' , -1 , pi_fr = (4./3.) , step = 0.4)
```

4.2 Plotting Scalar Data

After the scalar data has been converted there are two options for displaying the data using the VisIt package, automated and manual. The automated method which draws a Pseudocolor plot of the data, allows the user to modify the plot then renders an image sequence. The manual method requires the user to import the converted data into VisIt then after a plot has been setup a VisIt session saved. This session is used to import the settings of the VisIt plot to render the image sequence.

Once the user has finished rendering out the image sequence, the directory *batch* along with the VisIt session file can be removed. VisBOUTIt does not automatically remove the VTK files as they can be reused for generating new image sequences.

4.2.1 Automated method

The *draw* function, within the **draw.py** file, reads the converted data and creates a Psuedocolor plot of the data within a VisIt window. This allows the user to modify the view as desired (including changing the plot, slice operations, etc). The function will prompt for a session name of the VisIt view and for options regarding setting the maximum and minimum for the image sequence. The image sequence is then rendered to *dir_with_data/images/session_name/*. The filename has the format of: *VariableName_SessionName_batch_Time.PNG* , where the time part is a four digit number to ensure correct ordering. The arguments for the function are shown below.

```
draw(name , skip = 1 , path = None)
```

- **Name:** The variable name to be plotted, (string)
- **Skip:** The gap between the time slices. For example a skip value of 10 will convert the t = 0, 10, 20, etc slices. (Integer)

- **Path:** The path to the data, if the argument is not defined the script uses the working directory. (String)

An example of how to use the function is shown below. These commands import the draw functions and then draw the convert *'vort'* data within a VisIt session and then render out the image sequence.

```
$ from visboutit import draw
$ draw.draw('vort')
```

4.2.2 Manual method

This method allows the user to utilise the more advanced VisIt features than the automatic drawing method. This method uses the *image* function contained in the **image.py** file to render out the image sequence. The function has the same arguments as the draw function contained within **draw.py**. The script is designed to render out an image sequence of a VisIt session and the arguments are shown below.

```
image(name , skip = 1 , path = None)
```

First the converted data needs to be loaded and plotted in VisIt. This is achieved by launching VisIt from its install location and opening the converted data located in the *batch* folder. After customising the plot, a session file needs to be saved into the directory containing the raw BOUT++ data. The *image* function can then be executed to render out the image sequence. The script will prompt for; a session name and options for using fixed maximum and minimum values (only for Pseudocolor plots). Then render out the image sequence to the directory *data_directory/images/session_name/*. The session name should not include the file extension. An example of the functions use is shown below.

```
$ from visboutit import image
$ image.image('vort', path = '/path/to/data/dir')
Please Enter session name: vslice
Use max from max input file? (0 for No, 1 for Yes): 1
Use min from min input file? (0 for No, 1 for Yes): 1
```

The function will launch a VisIt session with the session loaded and render out the image sequence. The console will print messages from VisIt informing of which image has been written. After the image sequence has been rendered the VisIt window will close.

5 Eigen solver data

VisBOUTIt's eigen library displays the eigenvectors for a data set and allows the user to select a set of eigen values to convert to a specifed coordinate system. Before proceding ensure the initial setup has been performed detailed in section (3). Examples of how to plot data from BOUT++'s Eigen solver can be found within this section and within Appendix (C).

5.1 Converting Eigenvalues Data

To plot eigenvalues first the data needs to be converted to the desired coordinate system. The *draw* function within the *eigen.py* script handles the conversion. The *draw* function has the following arguments:

```
draw( name , path = None)
```

- **Name:** Name of the variable to be imported. (String)
- **path:** File path of the BOUT++ data, if it is not within the current working directory. (String)

The first step is to call the *draw* function which plots the eigen vectors real and imaginary components on a graph. When the function is called it will prompt for the coorindate system and coordinate system's settings. Then the function displays the eigenvectors on a graph. Second the user must select what values to convert by clicking on or near the points on the graph. There are different prompts for the various coordinate systems which are detailed below.

5.1.1 Cylinder Prompts

For the Cylinder coordinate system the *eigen.draw* function returns the below prompts.

```
$ from visboutit import eigen
$ eigen.draw('P')
Enter coordinate system; cylinder, torus or elm: cylinder
Enter step value for Interpolation (Default 0.5):
```

5.1.2 Torus Prompts

For the Torus coordinate system the *eigen.draw* function returns the below prompts.

```
$ from visboutit import eigen
$ eigen.draw('P')
Enter coordinate system; cylinder, torus or elm: torus
Enter step value for Interpolation (Default 0.5):
Use default specifications (y/n)?:
Use ISTTOK specifications R,r,dr,Bt,q = 0.46, 0.085, 0.02, 0.5, 5 (y/n):
```

If y is used for the *Use default specifications* option, this sets the variables in Equation (3) for the Torus model and procedes to drawing the eigenvector plot.

$$R,\ r,\ dr,\ Bt,\ q = \ 2.0,\ 0.2,\ 0.05,\ 1.0,\ 5.0 \tag{3}$$

If n is set for the defaults then the function prompts asking if the user would like to use the ISTTOK specifications. As shown in Equation (2).

$$R,\ r,\ dr,\ Bt,\ q = \ 0.46,\ 0.085,\ 0.02,\ 0.5,\ 5 \tag{2}$$

If n is selected for the ISTTOK specifications prompt then the user is prompted for the individual settings of torus mesh, as shown below.

```
Enter values for specifications
R =
r =
dr =
Bt =
q =
```

5.1.3 ELM Prompts

For the ELM coordinate system the eigen.draw function returns the below prompts.

```
$ from visboutit import eigen
$ eigen.draw('P')
Enter coordinate system; cylinder, torus or elm:
Enter zShift Interpolation Percent (Default 0.25) :
```

5.2 Plotting eigen Data

The eigen data that has been converted can be plotted automatically using the eigen function within the **draw.py** library. Or manually using VisIt then the image sequence can be rendered using the eigen function within the **image.py** library. Both methods create an image sequence of all of the converted data within the *batch* directory.

5.2.1 Automatic Method

The converted eigen data can be plotted using the eigen function within the **draw.py** library. The function has the following arguments.

```
eigen(name, path = None)
```

- **Name:** The name of the variable to be imported. (string)
- **Path:** The path that contains the raw data and the *batch* subdirectory. This is only required if the working directory does not contain those files. (string)

After the function has been called it will create two windows with the real eigenvalues in one and the imaginary eigenvalues in the second window. The function allows the user to modify the plot of both windows, however when the image sequence is rendered the orientation of the first will be applied to the second window. Please note that any operations need to be applied to both Windows ensuring both windows have the same Dimensions. If the windows do not have the same dimensions then the function will return an error.

After the session has been saved the image sequence is then rendered out to the directory: */images/SessionName/*. With the file format of: *VariableName_i_SessionName_image_time.png* . Where the i designates that the image is from imaginary data, an r in the place of the i designates that the image is from real data.

5.2.2 Manual Method

The manual method involes opening the data with VisIt, creating a VisIt session with a window for imaginary and another window for real data. After a VisIt session has been saved with the data plotted, then the images can be rendered using the eigen function within the **image.py** library. Which has the following arguments:

```
eigen(name, path = None)
```

- **Name:** The name of the variable to be imported, (string)
- **Path:** The path that contains the raw data and the *batch* subdirectory. This is only required if the working directory does not contain those files. (string)

After the function has been called it will prompt for the session name and then open the two VisIt windows and renders the image sequence. The image directory and filename have the same format as in the Automatic Method.
Directory: */images/SessionName/*
File Name format: *VariableName_i_SessionName_image_time.png*.

6 Vector data

The VisBOUTIt package can plot vectors that are parallel to the field lines of the coordinate system. For example flow velocity and parallel current density. Examples of how to plot vector data can be found within this section and within Appendix (B).

6.1 Converting Vector Data

The python library **vector.py** contains the functions; **vector.cylinder**, **vector.torus** and **vector.elm** functions that write the vectors VTK variables.

6.1.1 Cylinder

The cylinder function within the **vector.py** library converts the scalar data from BOUT++ data, for example parallel current density, into field parallel vectors. Which is achieved by creating an array of unit vectors from the mesh. At each point on the mesh a vector is created by performing a scalar dot product with the unit vector and the imported BOUT++ data. **vector.cylinder** function has the following arguments:

```
cylinder(name, time, pi_fr = (2./3.), step = 0.5 ,path = None, skip = 1)
```

- **Name:** variable name to import (string)
- **Time:** end time slice to convert to (integer), if -1 entire dataset is imported
- **pi_fr:** The fraction of π that the cylinder has been rotated through, i.e. 2π is a full cylinder. The default value for this is 2/3 . If the value given is above 2π then interger values of 2π are subtracted until the value is between zero and 2π then this value is used.(Float)

- **Step:** The gap between the y slices for the y interpolation, (float $0 \leq Step \leq 1$), if Step = 0 then the raw data displayed.
- **Path:** File path to data, if blank then the current working directory is used to look for data
- **Skip:** The gap between the time slices. For example a skip value of 10 will convert the t = 0, 10, 20, etc slices. (Integer)

6.1.2 Torus

The torus function within the **vector.py** library converts the scalar data from BOUT++ data, for example parallel current density, into field parallel vectors. Which is achieved by creating an array of unit vectors from the mesh. At each point on the mesh a vector is created by performing a scalar dot product with the unit vector and the imported BOUT++ data. **vector.torus** function has the below arguments. The arguments **Name, time, step, path** and **skip** are the same as in the Cylinder function.

```
torus(name, time, step = 0.5, path = None, R = None, r = None , dr = None ,
Bt = None, q = None , isttok = True, skip = 1)
```

- **R, r, dr, Bt, q:** Torus settings if left blank defaults in Equation (3) are used. R: Major radius r: Minor radius dr: Radial width of domain Bt: Toroidal magnetic field q: Safety factor. (All are floats)
- **isttok:** Use the ISTTOK specifications in Equation (2), (Boolean)

Once the fuction is called it will import, convert and write the vector variable to the VTK format within the *batch* directory. The example below will convert the variable *Vi* across all the data, with gaps of 20 in the time base.

```
$ from visboutit import vector
$ vector.torus('Vi', -1 , isttok = True , skip = 20)
```

6.1.3 ELM

The elm function within the **vector.py** library converts a scalar variable from BOUT++ data into field parallel vectors. Which is achieved by creating a mesh of unit vectors along the field lines. Then creates the vector at the points on the mesh by dotting the unit vector with the imported scalar value. The function **vector.elm** has the below arguments. The arguments **Name, time, path** and **skip** are the same as in the Cylinder and Torus functions.

```
elm(name , time ,zShf_int_p = 0.25, path = None, skip = 1)
```

- **zShf_int_p:** The zshift interpolation percentage (float, $0 \leq zShf\_int\_p \leq 1$) , zShf_int_p is 25% by default. This value is used to find the tolerance between the maximum and minimum zshift for linear interpolation in the y direction only. The larger the percentage the higher the tolerance value becomes therefore fewer interpolations are performed. The zshift tolerance is calculated by equation (1). The linear interpolation is performed when an array element set by the Pseudocode below is greater than zero. The integer determines how many points are to be inserted between the two original y slices.

```
int(abs( (zshift[y] - zshift[y+1] / z_tol ))
```

6.2 Plotting Vectors

Once the vector variables have been written to VTK format. The vectors can be plotted and image sequence rendered automatically or manually, the methods are detailed below.

6.2.1 Automatical Method

The automatic method uses the vector_draw function within the **draw.py** library. The function has the below arguments:

```
vector_draw(name , skip = 1, path = None)
```

- **Name:** Name of the variable (string)
- **Skip:** The skip value is the gap between the time values used in the conversion (integer)
- **Path:** File path to data, if blank then the current working directory is used. (String)

After the converted VTK files have been written to the *batch* directory. The vector_draw fuction is used to import the converted data and create a vector plot within a VisIt window. The plot can then be customised by the user and then a VisIt session saved using the prompt from the function. The function then renders the image sequence using the settings saved within the VisIt session. The image sequence is rendered into the directory: */dir/with/data/images/SessionName/*. With the file name of the format: *VariableName_SessionName_Time.png*. An example of the use of the code is shown below.

```
$ from visboutit import draw
$ draw.vector_draw('jpar' , skip = 5)
```

6.2.2 Manual Method

The manual method allows the user to utilise advanced features within VisIt. Which is achieved by loading the converted data, creating a plot and saving a session of the plot within VisIt. The vector function within the **image.py** library, is designed to render the image sequence of the VisIt session file. The function has the following arguments:

```
vector(name, skip = 1 , path = None)
```

- **Name:** The name of the variable to plot, (String)
- **Skip:** The skip value is the gap between the time values used in the conversion (integer)
- **Path:** File path to data, if blank then the current working directory is used. (String)

An example of how to use the image function is displayed below.

```
$ from visboutit import image
$ image.vector('jpar' , skip = 5)
Please Enter session name:
Use max from max input file? (0 for No, 1 for Yes):
Use min from min input file? (0 for No, 1 for Yes):
```

7 Acknowledgements

The author would like to acknowledge Dr B. Dudson for his work and guidance during the project. Also Michail Anastopoulos for his previous work on the project.

A Scalar Data Example

This section demonstrates how to import and plot scalar data outlined in Section (4).

1. Convert the raw BOUT++ data into the specified geometry using the relevant function within the **vtk.py** library. The command below imports the library and converts the selected variable from BOUT++ data across the entire data range with gaps of 20 in the timebase into Torus geometry.

   ```
   $ from visboutit import vtk
   $ vtk.torus('Vort' , -1 , skip = 20, isttok = True)
   ```

2. Create a VisIt session of the data either using the automatic method or manual method.

   (a) Automatic method uses the draw function within the **draw.py** library. The automatic method will render the image sequence after creating the VisIt window.

   ```
   $ from visboutit import draw
   $ draw.draw('Vort' , skip = 20)
   ```

   (b) Manually loading the data within VisIt and creating a session. After the session has been saved the image function within the **image.py** library can be used to render out an image sequence of the converted files. The code below will open a session and render out the image sequence.

   ```
   $ from visboutit import image
   $ image.image('Vort' , skip = 20)
   Please Enter session name:
   Use max from max input file? (0 for No, 1 for Yes):
   Use min from min input file? (0 for No, 1 for Yes):
   ```

3. The following figures show images from the different geometries.

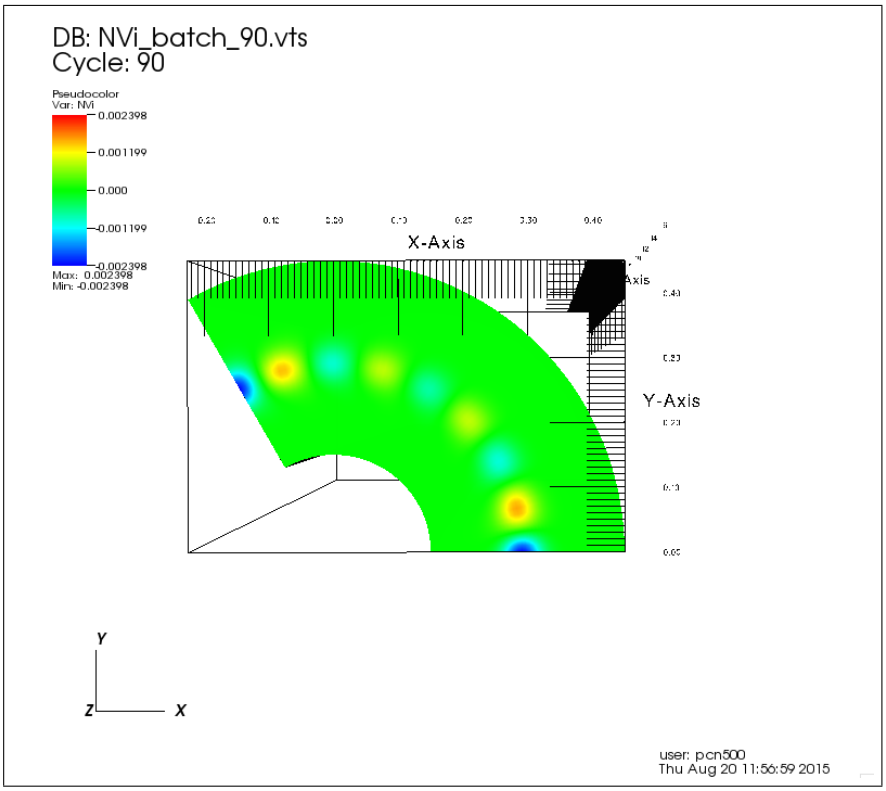

Figure 4: Cylinder Geometry plot of the converted scalar BOUT++ data.

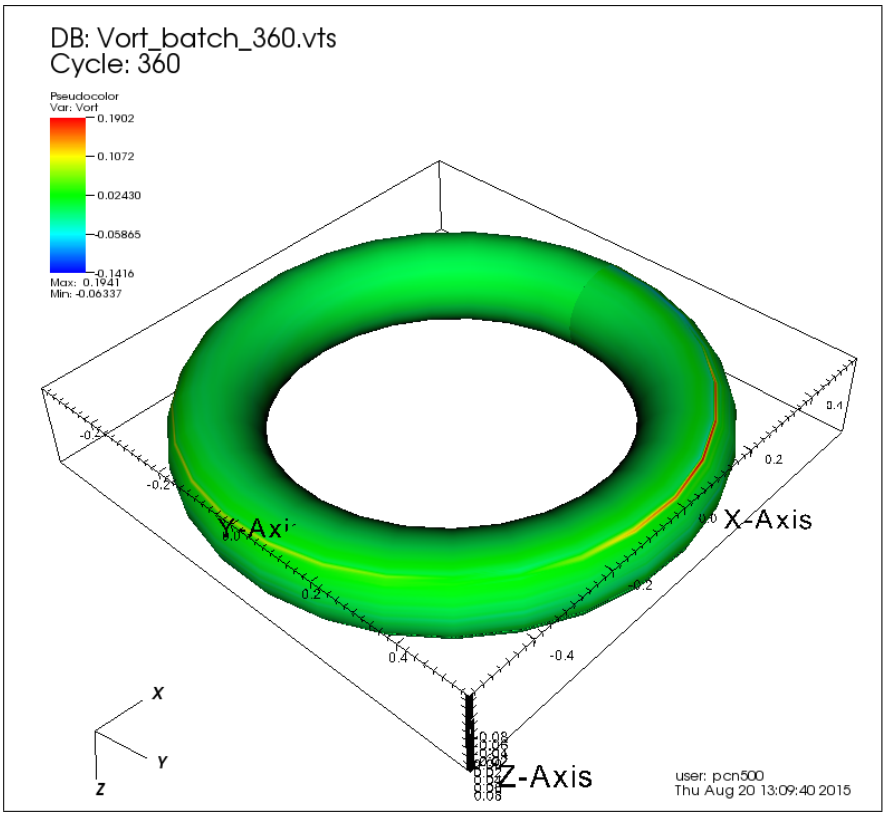

Figure 5: Torus Geometry plot of the converted scalar BOUT++ data.

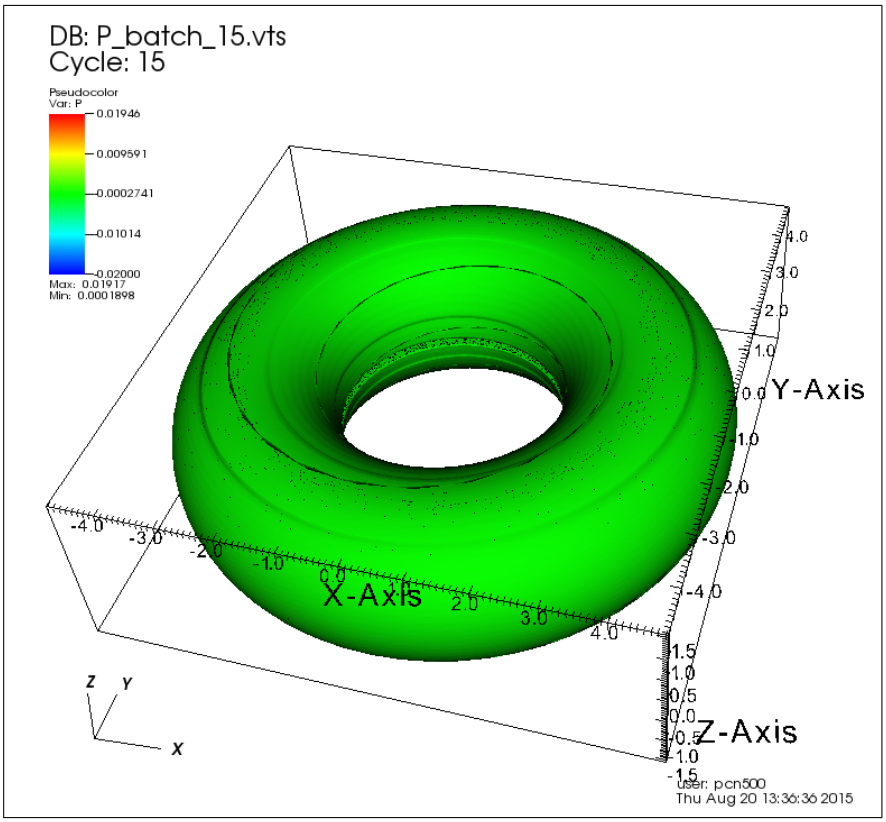

Figure 6: ELM Geometry plot of the converted scalar BOUT++ data.

B Field Parallel Vectors Data Example

Below is an example on how to use the cylinder function within the **vector.py** library to plot field parallel vectors from BOUT++ in cylindrical geometry.

1. Convert the scalar variable into a parallel field vector using the relevant function within **vector.py** library.

   (a) **Cylinder Geometry:** The *cylinder* function within the **vector.py** library is used to convert the coordinate system. The commands below will import a BOUT++ variable across the entire time range with gaps of 10 wihthin the time base and write the VTK files.

   ```
   $ from visboutit import vector
   $ vector.cylinder('NVi', -1, skip = 10)
   ```

   (b) **Torus Geometry:** The *torus* function within the **vector.py** library is used to convert the coordinate system. The commands below will import a BOUT++ variable across the entire time range with gaps of 20 wihthin the time base and write the VTK files.

   ```
   $ from visboutit import vector
   $ vector.torus('Vi', -1, skip = 20)
   ```

   (c) **ELM Geometry:** The *elm* function within the **vector.py** library is used to convert the coordinate system. The commands below will import a BOUT++

variable across the entire time range with gaps of 10 wihthin the time base and write the VTK files.

```
$ from visboutit import vector
$ vector.elm('jpar', -1, skip = 10)
```

2. Once the data has been converted a VisIt session needs to be created using the automatic or manual method

   (a) The Automatic method uses the *vector* function within the **draw.py** library to create a vector plot of the imported data within VisIt. This can be modified before saving a session. After saving a session the function will render out an image sequence of the VisIt window. The commands below create an VisIt session and image sequence of the converted data.

   ```
   $ from visboutit import draw
   $ draw.vector('jpar' , skip = 10)
   ```

   (b) The Manual method involes loading the converted data into VisIt and creating a session of a plot. After the session has been create the *vector* function within the **image.py** library can be used to render out an image sequence. The commands below open a VisIt session and render out the image sequence.

   ```
   $ from visboutit import image
   $ image.vector('jpar' , skip = 10)
   ```

The figures below display images of vector plots in the various geometries.

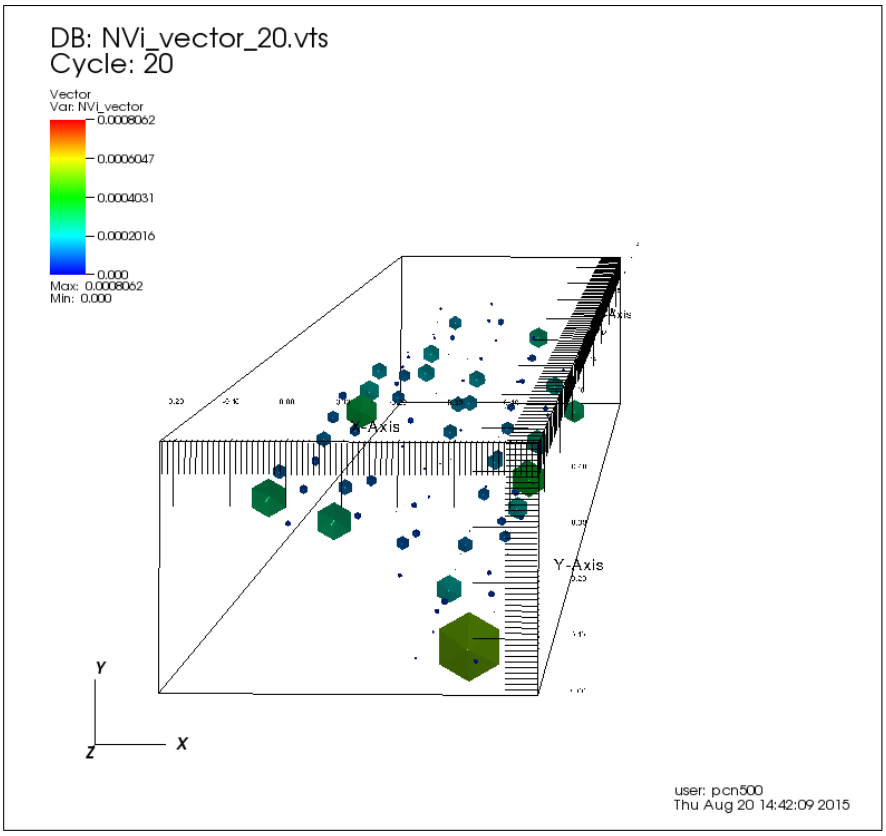

Figure 7: Cylinder Geometry plot of the converted field parallel vector BOUT++ data.

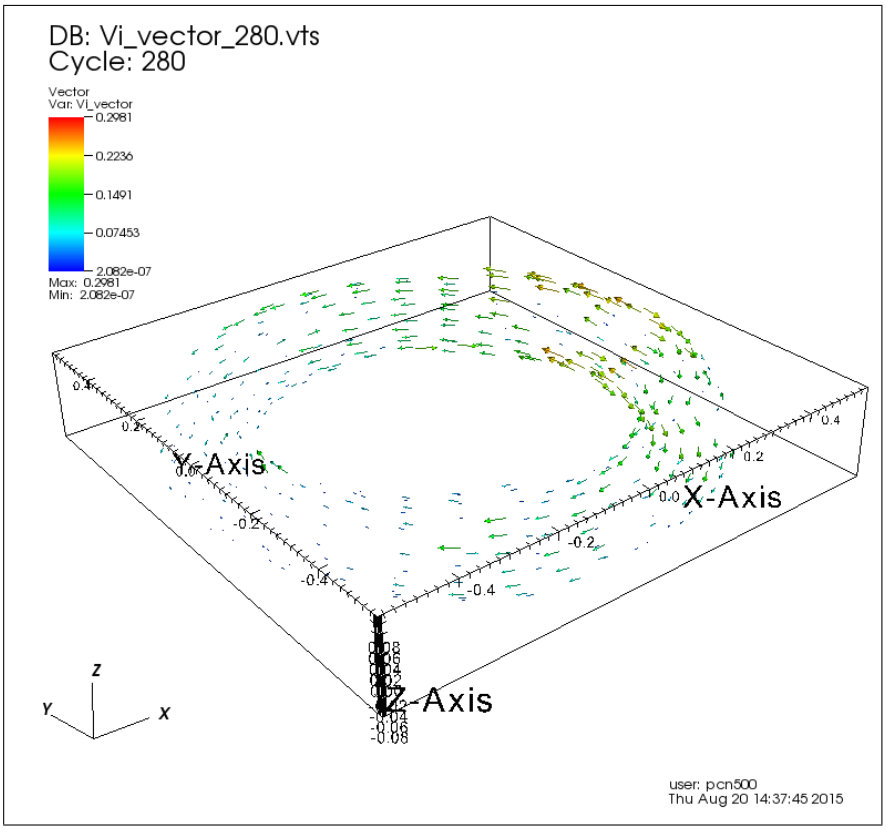

Figure 8: Torus Geometry plot of the converted field parallel vector BOUT++ data.

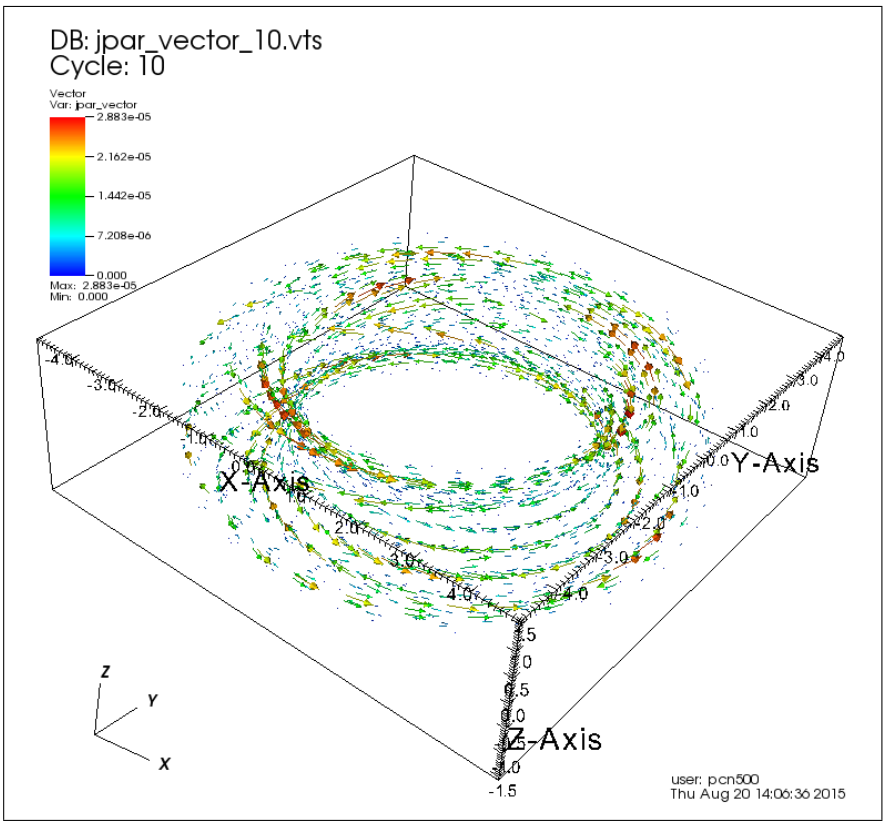

Figure 9: ELM Geometry plot of the converted field parallel vector BOUT++ data.

C Eigen Data Example

This section demonstrates how to import and plot scalar data from the BOUT++ eigensolver, as outlined in Section (5).

1. Call the function *draw* from the **eigen** library, with the variable name and path if required. Like below;

```
$ from visboutit import eigen
$ eigen.draw('P')
```

2. The function will prompt for the desired coordinate system the coordinate system's settings.

3. The function then displays the imported eigenvectors from the BOUT++ eigensolver, on a graph like that in Figure (10)

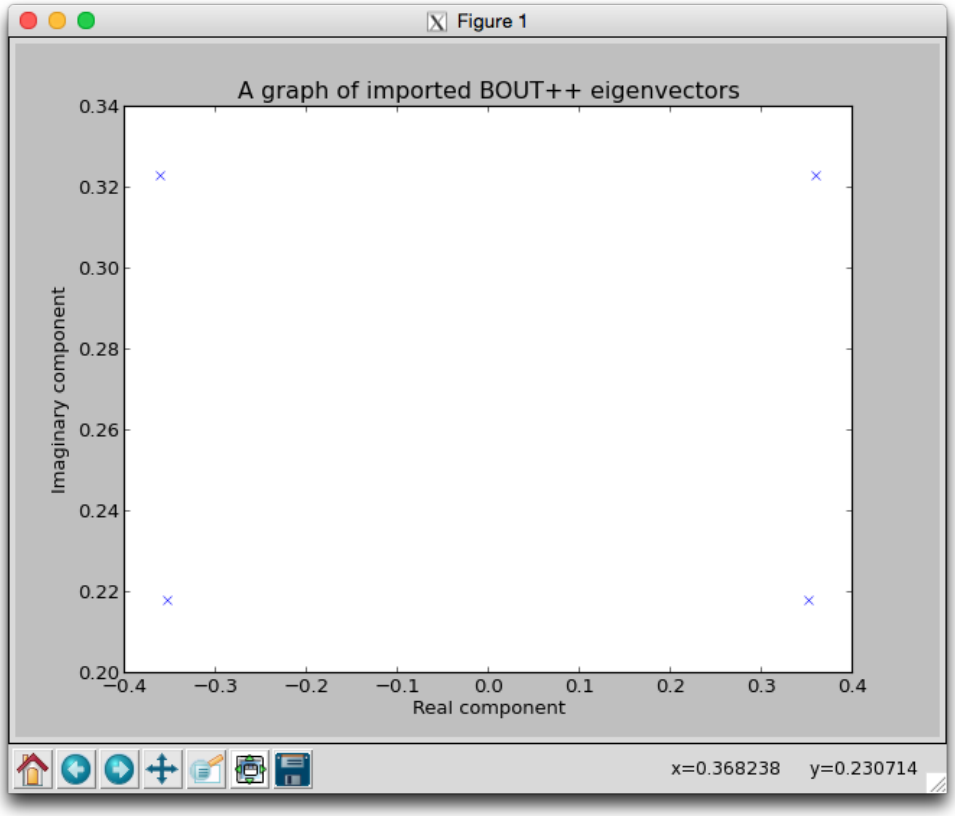

Figure 10: A Graph of the Eigenvectors imported from the BOUT++ eigensolver.

4. An eigenvector point needs to be clicked to convert a set of eigenvalues to the specified geometry.

5. After converting the data, a plot of the data can be created automatically or manually by the methods detailed below.

(a) **Automatic** The automatic method uses the function *eigen* within the **draw.py** library. This function creates two Pseudocolor plots of all of the available converted data, within the VisIt window. One window is for real values and the other is for imaginary values. The function allows the user to modify the plot before saving the session of the VisIt window. After the VisIt session has been saved the images of the plots are then rendered out. The commands below call the function and render out the images:

```
$ from visboutit import draw
draw.eigen()
```

(b) **Manual** The manual method involves the user manually loading the data into two seperate VisIt windows and saving a session of the desired orientation of the plot. The function *eigen* within the **image.py** library will load the session and render out all the images avaliable.

6. Figure (13) is an example of the images that can be produced from the VisBOUTIt package.

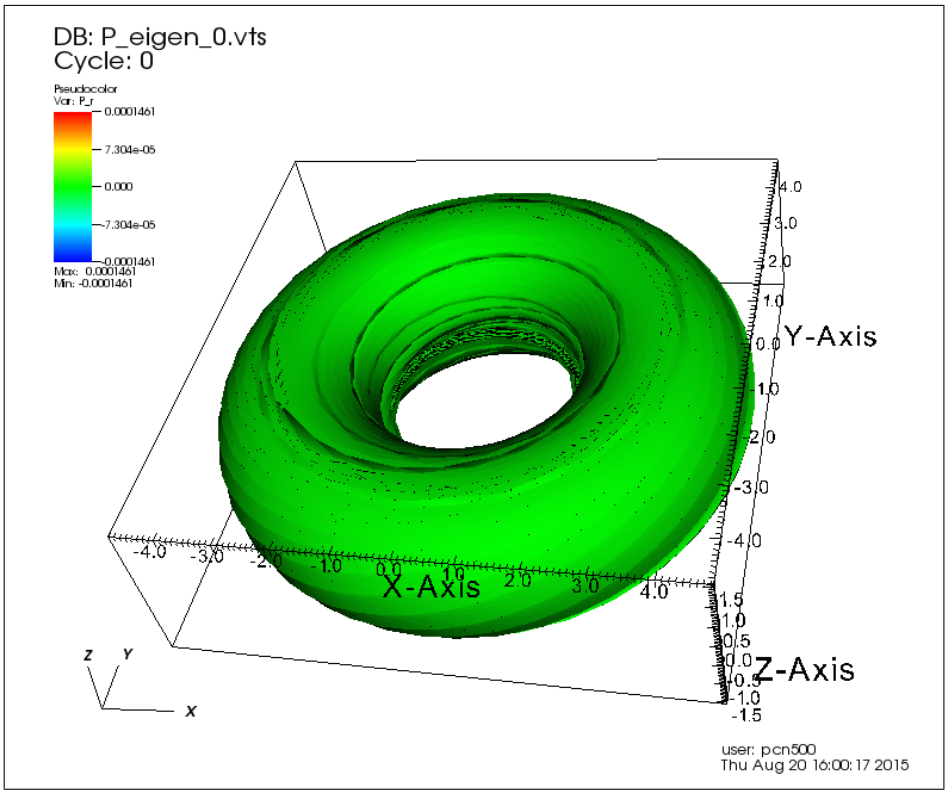

Figure 11: Real

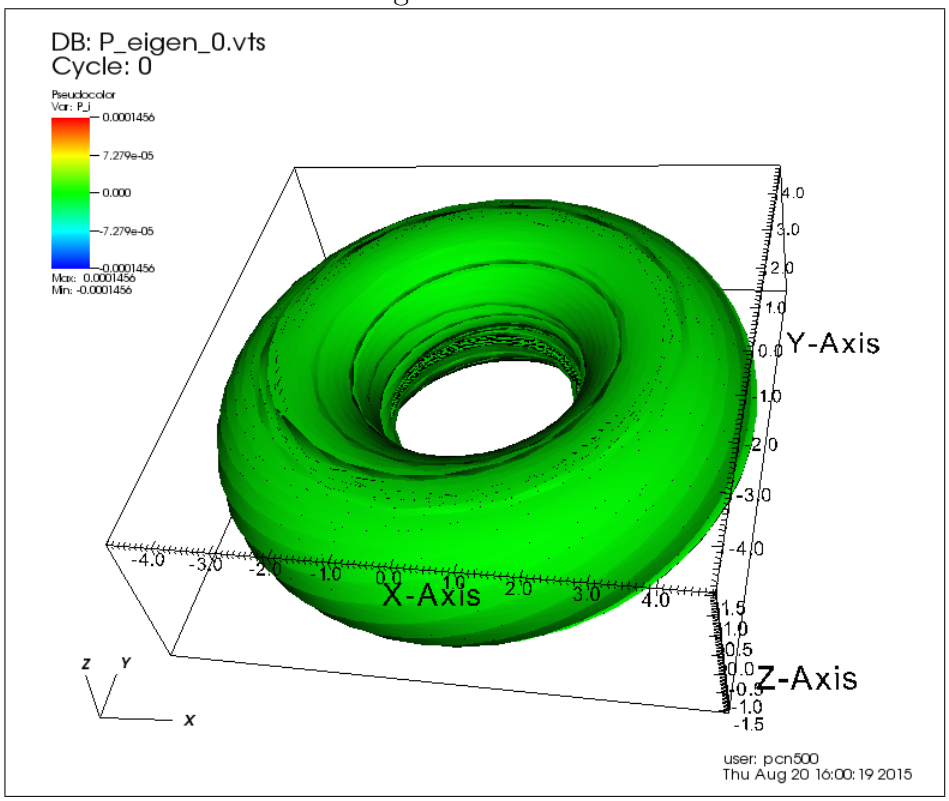

Figure 12: Imaginary

Figure 13: A plot of real Fig(11) and imaginary Fig(12) eigenvalues from the BOUT++ eigensolver