The TCV Psi–Toolbox tutorial

Contents

1. Introduction
   1. History and future
   2. Classes, objects and methods

2. Coordinate systems
   1. Coordinate systems: the psitbxgrid class
   2. How to manipulate psitbxgrid objects: some methods for that class
   3. When things do not align on a grid: the storage field
   4. When things move: the Time–Points storage
   5. Coordinate transformation: the psitbxg2g method
   6. One more step toward the tokamak: the toroidal coordinate system
   7. Parameter in coordinate transformation
   8. One more step toward variable configuration: moving magnetic axis
   9. Points in a time dependant coordinate system
   10. Time–Points in a time dependant coordinate system
   11. Now something simpler: 0–D, 1–D and 2–D coordinate systems
   12. Finally the ultimate complexity: N–D coordinate systems

3. Functions of the spatial coordinates and others
   1. Functions of the spatial position: the psitbxfun class
   2. Overloaded math operators for psitbxfun class
   3. Functions of the spatial position and of the time
   4. General forms from 0–D to N–D functions
   5. Function mapping: the psitbxf2f method
   6. Mapping of a function on Time–Points
   7. Mapping of a time dependant function on Time–Points
   8. A last extra: spatial derivative of functions
   9. Function integrals: the sum method
   10. Function average: the mean method

4. Equilibrium geometry description
   1. First description of the equilibrium: the psitbxpsi class
   2. Equilibrium description from TCV shot file
   3. Oh how easy it is now
   4. Normalised poloidal flux and magnetic axis: the psitbmxp method
   5. The Flux coordinate system
   6. Quantities depending only on the flux: profiles
   7. Coordinate transformation to flux coordinate system
   8. Flux surface description
   9. Flux surface description from the TCV shot file
   10. Coordinate transformation from flux coordinate system
   11. Flux coordinate metric
   12. Function integrals with metric
   13. Function average with metric
   14. Flux surface geometrical parameters

5. Diagnostic chords
   1. Diagnostic Chord Description: the psitbxcd class
   2. Moving chords
3. Diagnostic−Chords coordinate
4. Coordinate transformation from Diagnostic−Chords coordinates
5. Function mapping on a Diagnostic−Chords coordinates
6. Line integrals and averages
7. Transfer matrix and image reconstruction

<< |< >| Introduction

<< |< >| History and future

The Psi−Toolbox provides tools for manipulating space and time dependant quantities in the variable geometry of TCV. Targeted tasks are the analysis of the diagnostic data, analysis and simulation for transport and heating,... In summer 1995, extensive discussions with the potential users allowed to define the various wishes. Then in collaboration with Mathias Anton, we proposed a set of basic functionalities. In 1996, Christian Deschenaux coded a first version of these basic routines, which I later published as version 1. The result was pretty well described by Richard's words: I have spent the whole morning battling against the Psi−Toolbox. The fact is that the functionality was certainly adequat but the lack of detailed documentation made it unusable.

The advent of Matlab 5 encouraged me to completely revisit the Psi−Toolbox. Having written the first version, I realised that the various functionalities could be unified in a couple of simple concepts and the new capabilities of Matlab 5 would make the use of these concepts very clear. So users, if any, of the first Psi−Toolbox should forget their old coding habits and only retain what they want to achieve before starting reading this tutorial.

Although as you will see the concepts are very simple, the number of possibilities and combinations they generate is huge. So please let me know if you feel at some point that this tutorial is not clear enough. I will try to find better or more explicate examples. Also be tolerant with the numerous bugs you certainly will discover.

Ψ−Toolbox

Staring Richard A. Pitts
Thanks to Basil for giffing the figures

<< |< >| Classes, objects and methods

Version 2.0 of the TCV Psi−Toolbox intensively uses the capability of Matlab 5 for object oriented programming. Thus to better understand how to use this toolbox and how to get profit from it, you need to acquire a few concepts.

A class is a new data type reserved for a specific purpose. An object is an instance of a class. The first main step in the definition of a class is the choice of its data structure. In Matlab, objects are stored in structures whose fields contain the various information specific to the class. However these fields are neither visible nor accessible for the user. Instead, together with a class, methods are created which in principle safely manipulate these fields to perform operations specific to the class. One of these methods, the most fundamental, is the class constructor which creates the corresponding objects. In Matlab, the methods for a given class are gathered in a directory whose name is the class name preceeded by the string $S8A and the constructor is the method with the same name as the class.

Some basic methods, such as size or ndims, may apply to different classes. In such cases Matlab calls the appropriate method on the basis of the class of its arguments. Such methods are called overloaded.
A class can also be the child or a parent class. In that case, the child class inherits the fields and methods of its parent, to which specific fields and/or methods are added. It means that if a method is not defined for the child but only for the parent, the parent method will be used.

This is almost all what you need to know to use the Psi-Toolbox but if necessary further details can be found in chapter 14 of *Using Matlab*.

### Coordinate systems

### Coordinate systems: the psitbxgrid class

The first basic tools to take out from the Psi-Toolbox are the different coordinate systems. The information defining the systems you will use are stored in objects of class *psitbxgrid*. There are different coordinate systems and we will start with the 'Cylindrical' coordinates (r,z,phi) whose axis corresponds to the main vertical axis of the tokamak. Suppose Richard, an edge physicist, wants to study in details what happens around an inner wall tile. It is certainly a good idea for him to define a spatial grid in cylindrical coordinates around his tile, let say from 0.62 to 0.7 m in radial direction, from −0.05 to 0.05 m in vertical direction and from −3 to 3 degrees in toroidal angle. Thus he needs to construct an object of class *psitbxgrid*.

```matlab
rzphi = [0.62:.02:0.7,−.05:.02:.05,pi/180*(-3:2:3)];
GridCyl = psitbxgrid('Cylindrical','Grid',rzphi)
```

```matlab
GridCyl is a PsiTbx-Grid object =
  type: 'Cylindrical'
  label: {1x3 cell }
  storage: 'Grid'
  x: {1x3 cell }
  t: []
  par: [1x1 struct]
  csp: [1x1 struct]
```

```matlab
plot(GridCyl)
```
How to manipulate psitbxgrid objects: some methods for that class

You see that Matlab displays for Richard the fields of the psitbxgrid object and the corresponding contents. To use these fields, access with the usual dot syntax is defined. However to protect the integrity of the object, Richard is not allowed to directly modify them. The different methods applying on a psitbxgrid object will do that safely.

For example the field \( x \) is a cell array, each cell being the vector of the mesh points in each spatial direction. I will discuss the other fields later.

The function \( \text{size} \) has been redefined to give the effective size of the grid defined by the psitbxgrid object. It is a method of the psitbxgrid class.

```plaintext
GridCyl.label
ans =
    'R'    'Z'    'Phi'
GridCyl.x
ans =
[1x5 double]    [1x6 double]    [1x4 double]
GridCyl.x{1}, GridCyl.x{2}, GridCyl.x{3}
ans =
    0.6200    0.6400    0.6600    0.6800    0.7000
ans =
   -0.0500   -0.0300   -0.0100    0.0100    0.0300    0.0500
ans =
   -0.0524   -0.0175    0.0175    0.0524
size(GridCyl)
ans =
    5    6    4
which size -all
size is a built-in function.
```

When things do not align on a grid: the storage field

Richard has also installed on his tile two sets of ten probes of some kind. Of course these do not lie on a regular grid as the previously defined abstract points. So their position is specified by a triplet of coordinates, named 'Points'. It is the meaning of the storage field of a psitbxgrid object. In such a case the organisation of each cell can be arbitrary, for example here 2 by 10 matrices to represent the two sets of two probes. In addition, the numbers come from the drawing workshop and are specified on a cartesian 'Orthogonal' system, whose z-axis coincides with the machine main axis and whose x–z plane is at phi = 0.
xyz = (repmat(.625,2,10),[zeros(1,10);linspace(-.01,.01,10)],[linspace(-.01,.01,10);zeros(1,10)]);
ProbePointsOrt = psitbxgrid('Orthogonal','Points',xyz)
ProbePointsOrt is a PsiTbx-Grid object =
  type: 'Orthogonal'
  label: {1x3 cell  }
  storage: 'Points'
    x: {1x3 cell  }
    t: []
    par: [1x1 struct]
    csp: [1x1 struct]

plot(GridCyl), hold on, plot(ProbePointsOrt,'ob'), hold off

<< |< >| When things move: the Time-Points storage

Richard will tell you that things are never so simple. In fact his probes move in time. So the position of Richard's probe should be stored in multi-dimensional arrays whose last index is the time index. In this example each three spatial coordinate should be a 2 by 10 by number of times array. Then Richard must build a psitbxgrid object with 'Time-Points' storage:

vx = .1;
t = [.1,.3,.5];
dim = [ones(1,ndims(xyz{1})),length(t)];
xyzt = (repmat(xyz{1},dim)+vx*repmat(reshape(t,dim),[size(xyz{1}),1]),repmat(xyz{2},dim),repmat(xyz{3},dim));
MovingProbePointsOrt = psitbxgrid('Orthogonal','Time-Points',xyzt,t)
MovingProbePointsOrt is a PsiTbx-Grid object =
  type: 'Orthogonal'
  label: {1x3 cell  }
  storage: 'Time-Points'
    x: {1x3 cell  }
    t: [0.1000 0.3000 0.5000]
    par: [1x1 struct]
    csp: [1x1 struct]

plot(GridCyl), hold on, plot(MovingProbePointsOrt,'ob'), hold off
Coordinate transformation: the psitbxg2g method

Of course the first thing Richard wants to know is: where are my probes in cylindrical coordinates. The arithmetic is to be found in text book. The receipe to be used is the psitbxg2g method, which will hopefully do the job for him.

```matlab
ProbePointsCyl = psitbxg2g(ProbePointsOrt,'Cylindrical')
```

```plaintext
ProbePointsCyl is a PsiTbx-Grid object =
  type: 'Cylindrical'
  label: {1x3 cell }
  storage: 'Points'
    x: {1x3 cell }
    t: []
    par: [1x1 struct]
    csp: [1x1 struct]
ProbePointsCyl.x{1},ProbePointsCyl.x{2},ProbePointsCyl.x{3}
ans =
Columns 1 through 7
  0.6250    0.6250    0.6250    0.6250    0.6250    0.6250    0.6250
  0.6250    0.6250    0.6250    0.6250    0.6250    0.6250    0.6250
Columns 8 through 10
  0.6250    0.6250    0.6250
  0.6250    0.6250    0.6251
ans =
Columns 1 through 7
 -0.0100   -0.0078   -0.0056   -0.0033   -0.0011    0.0011    0.0033
  0         0         0         0         0         0         0
Columns 8 through 10
  0.0056    0.0078    0.0100
  0         0         0
ans =
Columns 1 through 7
  0         0         0         0         0         0         0
 -0.0160   -0.0124   -0.0089   -0.0053   -0.0018    0.0018    0.0053
Columns 8 through 10
  0         0         0
  0.0089    0.0124    0.0160
```
One more step toward the tokamak: the toroidal coordinate system

Toroidal coordinates is quite natural for a toroidal device. To transform orthogonal cartesian or cylindrical coordinates to toroidal coordinates, Richard must specify the position of the toidoidal axis. This is achieved with an additional argument to the \texttt{psitbxgrid} constructor. This argument can take several form. The one shown here is a cell array with the \((r,z)\) position of the axis. This information is then stored in the \texttt{par} field of the \texttt{psitbxgrid} object.

\begin{verbatim}
athetaphi = (linspace(0,.2,6),linspace(-pi,pi,7),linspace(0,2*pi,25));
r0z0 = (1,0);
GridTor = psitbxgrid(’Toroidal’,’Grid’,athetaphi,r0z0)
\end{verbatim}

\texttt{GridTor} is a \texttt{PsiTbx-Grid} object =
- type: ’Toroidal’
- label: \{1x3 cell \}
- storage: ’Grid’
- \(x\): \{1x3 cell \}
- \(t\): []
- \texttt{par}: \{1x1 struct\}
- \texttt{csp}: \{1x1 struct\}

\texttt{GridTor.par}
\texttt{ans =}
\[
\begin{array}{lll}
\texttt{r0: } & 1 \\
\texttt{z0: } & 0 \\
\texttt{psi: } & [] \\
\texttt{fsd: } & [] \\
\texttt{dcd: } & [] \\
\end{array}
\]

\begin{verbatim}
plot(GridTor)
\end{verbatim}

Parameter in coordinate transformation

The same kind of additional parameter may be necessary for transformation to toroidal coordinate. In this example a \texttt{psitbxgrid} object with toroidal coordinates is used, which contains in its \texttt{par} field the necessary information:

\begin{verbatim}
MovingProbePointsTor = psitbxg2g(MovingProbePointsOrt,’Toroidal’,GridTor)
\end{verbatim}

\texttt{MovingProbePointsTor} is a \texttt{PsiTbx-Grid} object =
- type: ’Toroidal’
One more step toward variable configuration: moving magnetic axis

In TCV configuration varies in time, in particular the magnetic axis moves. In the Psi–Toolbox, this motion can be taken into account with a time vector for the radial and vertical position of the axis. In this case the coordinate system changes at each time step. Note for ECH people that relativistic effect are not included.

\[ r_{0z0} = \{\text{ones}(1, \text{length}(t)), t \times 10^{-1}\} \]

\[ \text{MovingGridTor} = \text{psitbxgrid('Toroidal', 'Grid', \text{athetaphi}, r_{0z0}, t) } \]

MovingGridTor is a PsiTbx-Grid object =
- type: 'Toroidal'
- label: \{1x3 cell \}
- storage: 'Grid'
- x: \{1x3 cell \}
- t: \{0.1000 0.3000 0.5000\}
- par: \{1x1 struct\}
- csp: \{1x1 struct\}

plot(MovingGridTor)

Points in a time dependant coordinate system

However relativity tells us that even a static point can have varying coordinates if the coordinate system moves. So points stored as 'Points' must be transformed in 'Time-Points'. Note that in this example Richard has used directly the \text{psitbxgrid} object \text{GridTor} instead of the axis radial and vertical positions.

\[ \text{ProbePointsTor} = \text{psitbxg2g(ProbePointsOrt, 'Toroidal', GridTor) } \]

ProbePointsTor is a PsiTbx-Grid object =
- type: 'Toroidal'
- label: \{1x3 cell \}
Time–Points in a time dependant coordinate system

If your points are already stored as 'Time–Points', transformation to a time dependant coordinate system can be achieved only if the number of points in time are the same for both. The rule is that the Psi–Toolbox only handles a unique time base.

MovingProbePointsTor = psitbxg2g(MovingProbePointsOrt,'Toroidal',GridTor)
MovingProbePointsTor is a PsiTbx–Grid object =
  type: 'Toroidal'
  label: {1x3 cell }
  storage: 'Time–Points'
  x: {1x3 cell }
  t: [0.1000 0.3000 0.5000]
  par: [1x1 struct]
  csp: [1x1 struct]

Now something simpler: 0–D, 1–D and 2–D coordinate systems

Now that the difficult situation have been treated, we can come back to simpler things. Very often, the quantity you are dealing with does not depend on all spatial coordinates. For this purpose you can place a NaN on the implied spatial direction:

rzphi = {0.62:.02:0.7,−.05:.02:.05,NaN};
Grid2DCyl = psitbxgrid('Cylindrical','Grid',rzphi)
Grid2DCyl is a PsiTbx–Grid object =
  type: 'Cylindrical'
  label: {1x3 cell }
  storage: 'Grid'
  x: {1x3 cell }
  t: []
  par: [1x1 struct]
  csp: [1x1 struct]
plot(Grid2DCyl)
Having gone from 2D to 3D and then to time dependant coordinates, it was nothing for me to generalise. For example if Richard wants to study edge parameter fluctuation at different frequencies, he can define an additional dimension to his already complicated problem:

```
rzphifreq = [0.62:.02:0.7,-.05:.02:.05,pi/180*(-3:2:3),[1e4,2e4,5e4,1e5]];  
Grid4Dcyl = psitbxgrid('Cylindrical','Grid',rzphifreq);  
Grid4Dcyl is a PsiTbx-Grid object =  
  type: 'Cylindrical'  
  label: {1x3 cell  }  
  storage: 'Grid'  
     x: {1x4 cell  }  
     t: []  
    par: [1x1 struct]  
    csp: [1x1 struct]  
Grid4DOrt = psitbxg2g(Grid4Dcyl,'Orthogonal');  
Grid4DOrt is a PsiTbx-Grid object =  
  type: 'Orthogonal'  
  label: {1x3 cell  }  
  storage: 'Points'  
     x: {1x4 cell  }  
     t: []  
    par: [1x1 struct]  
    csp: [1x1 struct]  
Grid4DOrt.x{4}(1,1,1,:)  
ans(:,:,1,1) =  
  10000  
ans(:,:,1,2) =  
  20000  
ans(:,:,1,3) =  
  50000  
ans(:,:,1,4) =  
  100000
```
It is highly improbable that Richard will spend his life with coordinate systems. So it is time for him to open the Psi−Toolbox and take the \texttt{psitbxfun} class. This class will allow Richard to represent quantity that depend on the spatial position. Let us go back to the first grid Richard defined in cylindrical coordinates and suppose for simplicity that Richard's moving probes measure a given simple quantity. A \texttt{psitbxfun} object is then constructed by specifying the value of this function at the grid points and the grid where it is evaluated:

\begin{verbatim}
rzphi = GridCyl.x
rzphi =
    [1x5 double]   [1x6 double]   [1x4 double]
[rzphi(:)] = ndgrid(rzphi(:))
rzphi =
    [5x6x4 double]   [5x6x4 double]   [5x6x4 double]
x = exp(-300*(rzphi(1) - GridCyl.x{1}(1)).^2).*exp(-3000*rzphi(2).^2);
FunCyl = psitbxfun(x,GridCyl)
FunCyl is a PsiTbx−Function object =
    x: [5x6x4 double ]
    grid: [5x6x4 psitbxgrid]
    t: []
plot(FunCyl)
\end{verbatim}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{grid_cylinder.png}
\caption{Psi−Toolbox−Function object “FunCyl” (Cylindrical,Grid)}
\end{figure}

Almost all applicable mathematical operators\texttt{+−*\, / \, .* \, ./ \, .\, ’}, relational operators\texttt{== ~= > < >= <=}, logical operators\texttt{~ & |}, mathematical function\texttt{sin cos} \ldots are overloaded for \texttt{psitbxfun} objects. Experience:

\begin{verbatim}
plot(log(10*FunCyl))
\end{verbatim}
Functions of the spatial position and of the time

Quantities that varies also in time can be represented by 4-D arrays whos first three dimensions correspond to the spatial coordinates and the last one to time samples. The time array is then also passed to the `psitbxfun` constructor and stored in the appropriate field. Of course the last dimension and the length of the time vector must agree.

```
rzphi = GridCyl.x
rzphi = ...
[rzphi{:},t] = ndgrid(rzphi{:})
[rzphi{:}] = [1x5 double] [1x6 double] [1x4 double]
rzphit = {rzphi{:},t};
[rzphit{:}] = ndgrid(rzphit{:})
rzphit = ...
[rzphit{:}] = [4-D double] [4-D double] [4-D double] [4-D double]
x = exp(-300*(rzphit{1} - GridCyl.x{1}(1)).^2).*exp(-3000*rzphit{2}.^2).*exp(3*rzphit{4});
FunTimeCyl = psitbxfun(x,GridCyl,t)
```

General forms from 0-D to N-D functions

As for the grid, generalisation is straightforward. On an N-D grid `GridND` a function can have the size `[size(GridND),np1,np2,...,[,nt]]`. However I have no real example available yet. In addition if one of the grid coordinate is degenerated, being NaN or scalar, the corresponding singleton in the function value can be omitted. One common example are functions on a 2-D grid:

```
rz = Grid2DCyl.x
rz = ...
[rz{:}] = ndgrid(rz{:})
[rz{:}] = [1x5 double] [1x6 double] [NaN]
[rz{:}] = ndgrid(rz{:})
rz = ...
[rz{:}] = [5x6 double] [5x6 double] [NaN]
x = ((rz{1} - mean(Grid2DCyl.x{1})).^2 + (rz{2} - mean(Grid2DCyl.x{2})).^2);
Fun2DCyl = psitbxfun(x,Grid2DCyl)
```
Fun2DCyl is a PsiTbx-Function object =
  x: [5x6 double ]
  grid: [5x6x1 psitbxgrid]
  t: []

Function mapping: the psitbx2f method

Now this obvious question is: what does Richard measures with his probes. This can easily be answered using the psitbx2f method: the quantity defined in cylindrical coordinates must be estimated at the position of the probes. Let us start with fixed probes and a quantity constant in time:

FunProbePointsOrt = psitbx2f(FunCyl,ProbePointsOrt)
FunProbePointsOrt is a PsiTbx-Function object =
  x: [2x10 double ]
  grid: [2x10 psitbxgrid]
  t: []
FunProbePointsOrt.x
ans =
  Columns 1 through 7
     0.7370    0.7826    0.8169    0.8397    0.8511    0.8511    0.8397
          0.8524    0.8524    0.8525    0.8525    0.8526    0.8526    0.8525
  Columns 8 through 10
     0.8169    0.7826    0.7370
          0.8525    0.8524    0.8524

Mapping of a function on Time-Points

Now a quantity constant in time measured by moving probes:

FunMovingProbeOrt = psitbx2f(FunCyl,MovingProbePointsOrt)
FunMovingProbeOrt is a PsiTbx-Function object =
  x: [2x10x3 double ]
  grid: [2x10x3 psitbxgrid]
  t: [0.1000 0.3000 0.5000]
plot(FunMovingProbeOrt,'o','markersize',12)
Mapping of a time dependant function on Time–Points

Finally a time dependant quantity measured by moving probes:

```
FunTimeMovingProbeOrt = psitbxf2f(FunTimeCyl,MovingProbePointsOrt)
FunTimeMovingProbeOrt is a PsiTbx-Function object =
    x: [2x10x3 double    ]
    grid: [2x10x3 psitbxgrid]
    t: [0.1000 0.3000 0.5000]
plot(FunTimeMovingProbeOrt,'o','markersize',12)
```

A last extra: spatial derivative of functions

The `psitbxf2f` method can also be used to compute the spatial derivative of functions, by specifying the derivation order along each spatial coordinate. Note that since functions are interpolated by cubic spline, the third derivative is the last non zero derivatives.

```
DerDRFunTimeMovingProbeOrt = psitbxf2f(FunTimeCyl,MovingProbePointsOrt,[1,0,0])
DerDRFunTimeMovingProbeOrt is a PsiTbx-Function object =
    x: [2x10x3 double    ]
    grid: [2x10x3 psitbxgrid]
    t: [0.1000 0.3000 0.5000]
plot(DerDRFunTimeMovingProbeOrt,'o','markersize',12)
```
Function integrals: the sum method

Integration along the definition variables of functions defined on a 'Grid' is the task of the sum method. Summation along a dimension corresponding to one of the grid coordinates or the time results in an integration. Along the other dimensions, it is a simple summation. The dimension argument can be 't' with obvious effect. As an example take the `psitbxtcv` function with mode 'JPHI', which will return a `psitbxfun` object containing the toroidal current distribution.

```matlab
jphi = psitbxtcv(10000,t,'JPHI')
jphi is a PsiTbx-Function object =
  x: [28x65x3 double  ]
  grid: [28x65x1 psitbxgrid]
  t: [0.1000 0.3000 0.5000]
plot(jphi)
```
Certainly the most non-trivial way to obtain the plasma current is:

\[ Ip = \text{sum}(jphi,[1,2]) \]

\( Ip \) is a PsiTbx-Function object =

\[
\begin{align*}
x & : [1.9170e+05 \ 2.8147e+05 \ 2.8277e+05] \\
grid & : [1x1x1 \ \text{psitbxgrid}] \\
t & : [0.1000 \ 0.3000 \ 0.5000] \\
\end{align*}
\]

\[ Ip.grid.x \]
\[ \text{ans} = \]
\[ [\text{NaN}] \]
\[ [\text{NaN}] \]
\[ [\text{NaN}] \]

\[ \text{plot}(Ip) \]

---

<< |< >| Function average: the mean method

Average is the task of the \texttt{mean} method, which works more or less like \texttt{sum}. However it accepts an optional argument, a \texttt{psitbxfun} object which will play the role of a weighting function. In fact in that case \texttt{mean} is simply

\[ \text{sum}(x.*w)./\text{sum}(w) \]:

\[ javg = \text{mean}(jphi,[1,2],jphi>0) \]

\( javg \) is a PsiTbx-Function object =

\[
\begin{align*}
x & : [7.5580e+05 \ 9.0686e+05 \ 9.7908e+05] \\
grid & : [1x1x1 \ \text{psitbxgrid}] \\
t & : [0.1000 \ 0.3000 \ 0.5000] \\
\end{align*}
\]

\[ \text{plot}(javg) \]
Equilibrium geometry description

Tokamak equilibrium is very often described with the spatial distribution of the poloidal flux in cylindrical coordinate. Of course since Tokamak is axisymmetric, this flux does not depend on the axial coordinate.

First description of the equilibrium: the psitbxpsi class

It is pretty obvious that the poloidal flux used to describe the Tokamak equilibrium could be stored in a psitbxfun object with cylindrical coordinates. However to add some functionality I defined a new class, the psitbxpsi class, child of the psitbxfun class. This means that all methods defined for the psitbxfun class apply to the psitbxpsi class and that a psitbxpsi object contains all the fields of a psitbxfun object plus others. The first additional field is the format field which specifies the way the poloidal flux is stored. It is a two character string each with possible values of ‘−’, ‘0’ or ‘+’. The first character gives the sign of the poloidal flux in the plasma center, the second on the plasma edge. A useless example of a psitbxpsi object would be:

```plaintext
psi = psitbxpsi(ndgrid(Grid2DCyl.x{1},Grid2DCyl.x{2}),Grid2DCyl,[],'−0')
```

psi is a PsiTbx-Poloidal-Flux object =
format: '−0'
rmag: []
zmag: []
psitbxfun: [5x6 psitbxfun]
iter: 10
tol: 1.0000e-04

Equilibrium description from TCV shot file

Much more useful is the description of a real TCV shot. This can be easily built with the psitbxtcv function. Here Richard selected times in his favorite standard shot #10000:

```plaintext
psi = psitbxtcv(10000,t)
psi is a PsiTbx-Poloidal-Flux object =
format: '+0'
rmag: []
```
Oh how easy it is now

A long time ago Richard wanted to know the poloidal magnetic field near his tiles, basically given by the derivative of the poloidal flux along the radial and vertical directions. At this point you realise how easy it is:

\[
\text{drpsi} = \text{psitbxf2f}(\text{psi}, \text{Grid2DCyl}, [1 \ 0])
\]
\[\text{drpsi}\] is a \text{PsiTbx-Function} object =
\[
\begin{align*}
\text{x:} &\quad [5x6x3 \text{ double }] \\
\text{grid:} &\quad [5x6x1 \text{ psitbxgrid}] \\
\text{t:} &\quad [0.1000 \ 0.3000 \ 0.5000]
\end{align*}
\]
\[
\text{dzpsi} = \text{psitbxf2f}(\text{psi}, \text{Grid2DCyl}, [0 \ 1])
\]
\[\text{dzpsi}\] is a \text{PsiTbx-Function} object =
\[
\begin{align*}
\text{x:} &\quad [5x6x3 \text{ double }] \\
\text{grid:} &\quad [5x6x1 \text{ psitbxgrid}] \\
\text{t:} &\quad [0.1000 \ 0.3000 \ 0.5000]
\end{align*}
\]

\text{plot(drpsi)}
Normalised poloidal flux and magnetic axis: the psitbxp2p method

Most of the Psi–Toolbox functions requires normalised poloidal flux which will later serve as radial coordinate: A normalised poloidal flux is one on the last close flux surface and zero on magnetic axis. Normalisation is achieved with the method psitbxp2p on a psitbxpsi object. It locates the magnetic axis position stored in rmag and zmag fields and modifies the value of psi:

```matlab
psi = psitbxp2p(psi,'01')
psi is a PsiTbx-Poloidal-Flux object =
    format: '01'
    rmag: [0.8877 0.8941 0.8977]
    zmag: [0.2352 0.2450 0.2530]
    psitbxfun: [28x65x3 psitbxfun]
    iter: 10
    tol: 1.0000e-04
plot(psi)
```
Note that specifying the format in the `psitbxtcv` call would yield the same result:

```matlab
g = psitbxtcv(10000,t,'01')
g is a PsiTbx-Poloidal-Flux object =
   format: '01'
   rmag: [0.8877 0.8941 0.8977]
   zmag: [0.2352 0.2450 0.2530]
gpsitbx = [28x65x3 g]
   iter: 10
   tol: 1.0000e-04
```

<< | <| The Flux coordinate system

Let me go back to the coordinate systems and introduce the very important flux coordinate system. This is basically a toroidal coordinate system in which the distance from the axis is replaced by the square root of the normalised poloidal flux. Although not mandatory, it may be very useful to associate to such a coordinate system the value of the normalised poloidal flux stored in a `psitbxpsi` object:

```matlab
GridFluxPsi = psitbxgrid('Flux', 'Grid', 'Default', psi)
GridFluxPsi is a PsiTbx-Grid object =
   type: 'Flux-Coordinate'
   label: {1x3 cell  }
   storage: 'Grid'
   x: {1x3 cell  }
   t: [0.1000 0.3000 0.5000]
   par: [1x1 struct]
   csp: [1x1 struct]
size(GridFluxPsi)
ans =
   41  129  1
```

<< | <| Quantities depending only on the flux: profiles

A very common concept in Tokamak physics is the profile, that is a quantity constant on a flux surface and therefore depending only on the flux coordinate. Such a profile is trivially represented by a function on a 1-D grid, here the
canonical parabolic profile:

\[
\text{GRID1DRHO} = \text{PSITBXGRID('FLUX','GRID', \{linspace(0,1,41),NaN,NaN\}, psi)
}
\]

GRID1DRHO is a PsiTbx-Grid object =
  type: 'Flux-Coordinate'
  label: \{1x3 cell\}
  storage: 'Grid'
    x: \{1x3 cell\}
    t: \{0.1000 0.3000 0.5000\}
    par: \{1x1 struct\}
    csp: \{1x1 struct\}

\[
\text{PARABOLICRHO} = \text{PSITBXFUN(1-linspace(0,1,41)').^2,GRID1DRHO)
}\]

PARABOLICRHO is a PsiTbx-Function object =
  x: [41x1 double]
  grid: [41x1x1 psitbxgrid]
  t: []

plot(PARABOLICRHO)

<< | < > | Coordinate transformation to flux coordinate system

Of course discovering this new coordinate system, Richard wants to know where his moving probes lies in term of flux coordinate. The psitbxg2g method applies. Transformation from any coordinate system to flux coordinate requires a normalised poloidal flux description:

\[
\text{MOVINGPROBEPOINTSFLUX} = \text{PSITBXG2G(MOVINGPROBEPOINTSORT, 'FLUX', psi)
}\]

MOVINGPROBEPOINTSFLUX is a PsiTbx-Grid object =
  type: 'Flux-Coordinate'
  label: \{1x3 cell\}
  storage: 'Time-Points'
    x: \{1x3 cell\}
    t: \{0.1000 0.3000 0.5000\}
    par: \{1x1 struct\}
    csp: \{1x1 struct\}

MOVINGPROBEPOINTSFLUX.x(1)
ans(:,1,:) =
  Columns 1 through 7
  1.0544  1.0536  1.0529  1.0522  1.0514  1.0507  1.0499
An alternative but very important representation of the equilibrium geometry in the Psi−Toolbox is the description of the flux surfaces themselves. These are contours of constant poloidal flux; this is formalised as the distance from the magnetic axis as a function of the square root of the normalised flux and the poloidal angle. It can be stored in a psitbxfun object with a flux coordinate grid. But here once more to get additional functionality, it is stored in a psitbxpsi object with format 'FS'. Conversion from a psitbxpsi object containing the distribution of the poloidal flux is achieved with the psitbxp2p method:

```matlab
fsd = psitbxp2p(psi,'FS')
```

fsd is a PsiTbx−Poloidal−Flux object =
format: 'FS'
rmag: [0.8877 0.8941 0.8977]
zmag: [0.2352 0.2450 0.2530]
psitbxfun: [41x129x3 psitbxfun]
iter: 10
tol: 1.0000e−04
plot(fsd)

<< | >| Flux surface description

An alternative but very important representation of the equilibrium geometry in the Psi−Toolbox is the description of the flux surfaces themselves. These are contours of constant poloidal flux; this is formalised as the distance from the magnetic axis as a function of the square root of the normalised flux and the poloidal angle. It can be stored in a psitbxfun object with a flux coordinate grid. But here once more to get additional functionality, it is stored in a psitbxpsi object with format 'FS'. Conversion from a psitbxpsi object containing the distribution of the poloidal flux is achieved with the psitbxp2p method:

```matlab
fsd = psitbxp2p(psi,'FS')
```

fsd is a PsiTbx−Poloidal−Flux object =
format: 'FS'
rmag: [0.8877 0.8941 0.8977]
zmag: [0.2352 0.2450 0.2530]
psitbxfun: [41x129x3 psitbxfun]
iter: 10
tol: 1.0000e−04
plot(fsd)
Now try to catch this:

```matlab
plot(fsd.psitbxfun)
```

Flux surface description from the TCV shot file

Richard will very soon realise that the `psitbxp2p` method takes a lot of time. To spare Richard's time, the necessary information is stored in the TCV shot file once the equilibrium has been reconstructed. Thus a fast way to recover the flux surface contour for an old shot is to use the `psitbxtcv` function with format 'FS':

```matlab
fsd = psitbxtcv(10000,t,'FS')
fsd is a Psitbx-Poloidal-Flux object =
    format: 'FS'
    rmag: [0.8877 0.8941 0.8977]
```
Coordinate transformation from flux coordinate system

The first usage of a `psitbxpsi` object of format FS is for coordinate transformation from flux coordinates:

```matlab
rhothetaphi = ([0,.5,1], linspace(-pi,pi,9), linspace(0,pi/2,4));
pplot(psitbxg2g(psitbxgrid('Flux','Grid',rhothetaphi),'Orthogonal',fsd),'−or','markersize',12)
```

Flux coordinate metric

The main difficulty with flux coordinates is that it is not an orthogonal coordinate system, which renders its metric non-trivial. The Psi−Toolbox provides a method for taking the appropriate metric into account, simply called `metric`. It applies to a `psitbxgrid` object with 'Grid' storage or a `psitbxpsi` object with format 'FS' and takes as second argument the combination of dimension for which you need the metric. For example if you want the metric on the flux surface, use `[2,3]` to get the unit area on (theta,phi) planes; for unit volume, use `[1,2,3]`. You can also get the norm of the coordinate gradient with a corresponding negative dimension. In the example the very important gradient of rho entering the expression of the SEF (Shape Enhancement Factor) is plotted. This method returns a `psitbxfun` object and also works for coordinate systems with diagonal metric.

```matlab
GradRho = metric(fsd,-1)
GradRho is a PsiTbx-Function object =
x: [41x129x3 double]
grid: [41x129x1 psitbxgrid]
t: [0.1000 0.3000 0.5000]
pplot(GradRho)
```
Function integrals with metric

Using the `sum` method with the option `metric` allows to perform spatial integrals with the correct metric. In the example the toroidal current distribution in cylindrical coordinates is first mapped in flux coordinates and then surface averaged:

```plaintext
GridFluxFSD = psitbxgrid('Flux', 'Grid', 'Default', fsd)
GridFluxFSD is a PsiTbx-Grid object =
  type: 'Flux-Coordinate'
  label: {1x3 cell }
  storage: 'Grid'
    x: {1x3 cell }
    t: [0.1000 0.3000 0.5000]
    par: {1x1 struct}
    csp: {1x1 struct}

jphiFlux = psitbxf2f(jphi, GridFluxFSD)
jphiFlux is a PsiTbx-Function object =
  x: [41x129x3 double ]
  grid: [41x129x1 psitbxgrid]
  t: [0.1000 0.3000 0.5000]

IpFlux = sum(jphiFlux, [1, 2], 'metric')
IpFlux is a PsiTbx-Function object =
  x: [1.9163e+05 2.8115e+05 2.8208e+05]
  grid: [1x1x1 psitbxgrid]
  t: [0.1000 0.3000 0.5000]

plot(Ip), hold on, plot(IpFlux, '-og'), hold off
```
Function average with metric

The mean method can also be used with a 'metric' option. The simplified definitions are:

\[
\text{mean}(f, \text{dim}, w, '\text{metric}') = \frac{\sum(f.*w.*\text{metric}(f.\text{grid}, \text{dim}), \text{dim})}{\sum(w.*\text{metric}(f.\text{grid}, \text{dim}), \text{dim})}
\]

\[
\text{mean}(\text{jphiFlux}, [1,2], '\text{metric}')
\]
\text{ans is a PsiTbx-Function object =}
\[
\begin{align*}
x: [7.5539e+05 & 9.0218e+05 9.6204e+05] \\
grid: [1x1x1 \text{psitbxgrid}] \\
t: [0.1000 & 0.3000 0.5000]
\end{align*}
\]

\[
\text{mean}(\text{jphi}, [1,2], \text{jphi} > 0)
\]
\text{ans is a PsiTbx-Function object =}
\[
\begin{align*}
x: [7.5580e+05 & 9.0686e+05 9.7908e+05] \\
grid: [1x1x1 \text{psitbxgrid}] \\
t: [0.1000 & 0.3000 0.5000]
\end{align*}
\]

Flux surface geometrical parameters

Quantities such as the flux surface area, cross section, inside volume or the volume or cross-section between two adjacent flux surfaces are widely used. Also very important since it enters in the expression of the SEF is the flux surface average of the gradient of rho. They are now obvious to compute using the following definitions:

\[
grho \text{ flux surface average of the gradient of rho}
\]
\text{mean(metric(g,[-1]),[2,3])}

\[
surf \text{ flux surface area}
\]
\text{sum(metric(g,[2,3]),[2,3])}

\[
area \text{ flux surface cross-section}
\]
\text{cumsum(sum(metric(g,[1,2]),2),1)}

\[
vol \text{ flux surface inside volume}
\]
\text{cumsum(sum(metric(g,[1,2,3]),[2,3]),1)}

\[
darea \text{ cross-section between two flux surfaces divided by the flux coordinate increment}
\]
\text{cumsum(sum(metric(g,[1,2,3]),[2,3]),1)}
sum(metric(g,[1,2]),2) 
\textit{dvol volume between two flux surfaces divided by the flux coordinate increment} 
sum(metric(g,[1,2,3]),[2,3])

All these quantities are returned in a structure by the method \texttt{psitbxfsf}:

```python
fsg = psitbxfsf(fsd)
fsg =
    grho: [41x3 psitbxfun]
surf: [41x3 psitbxfun]
darea: [41x3 psitbxfun]
area: [41x3 psitbxfun]
dvol: [41x3 psitbxfun]
vol: [41x3 psitbxfun]
plot(fsg.grho)
```

\begin{center}
\includegraphics[width=\textwidth]{image.png}
\end{center}

\textbf{<< | < >| Diagnostic chords}

\textbf{<< | < >| Diagnostic Chord Description: the psitbxdc object class}

Describing the geometry of a multi–chord diagnostic require to gather some information. This will be arranged in a \texttt{psitbxdc object} with the following properties:

\begin{itemize}
  \item \texttt{rd} 
    \begin{itemize}
      \item The radial position in cylindrical coordinate of a reference point along the chord, very often the pin–hole
    \end{itemize}
  \item \texttt{zd} 
    \begin{itemize}
      \item The vertical position of the same point
    \end{itemize}
  \item \texttt{phid} 
    \begin{itemize}
      \item The toroidal angle of the same point
    \end{itemize}
  \item \texttt{pvd} 
\end{itemize}
The poloidal viewing angle, measured between the horizontal plane and the chord, positive if the chord is pointing above the plane

\[ tvd \]

The toroidal viewing angle, measured between the vertical plane containing the machine axis and the chord reference point and the vertical plane containing the chord itself, positive if the chord is looking in the direction of increasing toroidal angle

\[ nd \]

The number of points used to discretise the chord

\[ t \]

The time base for moving chords

Here is an example with two cameras with 20 detectors each, the first one with a poloidal coverage, the second one with a toroidal coverage. Note that the fields of a \texttt{psitbxdcd} object can retain an appropriate organisation, here 2 by 20, or 1028 by 1028 for CCD fans.

\begin{verbatim}
rd = 1.2;
zd = 0;
phid = pi/8;
pvd = [linspace(-pi/3,pi/3,20);zeros(1,20)];
tvd = [zeros(1,20);linspace(-pi/3,pi/3,20)];
dcd = psitbxdcd(rd,zd,phid,pvd,tvd);
plot(dcd)
\end{verbatim}

\section*{Moving chords}

The possibility to have real–time scanning diagnostic or beam is included. In that case the last dimension of the fields of the \texttt{psitbxdcd} object represents the time base. Given as an example \textit{Le cauchemar du gyrotroniste}:

\begin{verbatim}
td = linspace(0,1,73);
rd = 1.2;
\end{verbatim}
zd = 0;
phid = pi/2;
pvd = pi/6*cos(pi*td);
tvd = pi/6*sin(2*pi*td);
LeCauchemarDuGyrotroniste = psitbxcd(rd,zd,phid,pvd,tvd,[],td);
plot(LeCauchemarDuGyrotroniste)

<< | < > | Diagnostic−Chords coordinate

In essence a diagnostic chord is an object with only one dimension, defined as the distance from the reference point. Although certainly possible, I have not left the possibility to have a constant grid along all the chords. Only 'Points' or 'Time−Points' storage is possible. The size of the array representing the chord coordinate must be [dcd.nd, size(dcd.rd)]. Since it is somehow fastidious to chose a good grid for each chord, I have made the intersect method which returns the chord coordinate so that it covers the intersection of the chords and a cylindrical coordinate grid.

GridDiagChords = psitbxgrid('Diagnostic−Chords','Points',{intersect(dcd,psi.grid)},dcd)
GridDiagChords is a PsiTbx−Grid object =
  type: 'Diagnostic−Chords'
  label: {1x1 cell }
  storage: 'Points'
    x: {1x1 cell }
    t: []
    par: [1x1 struct]
    csp: [1x1 struct]
plot(psitbxgrid('Cylindrical','Grid',{{.614,1.147},[-.76,.76],linspace(-pi/2,pi,10)}))
hold on, plot(GridDiagChords,'−b'), hold off
Coordinate transformation from Diagnostic–Chords coordinates

Any coordinate transformation from Diagnostic–Chords coordinate is possible. The inverse is not possible since chords are discrete and do not cover the whole space.

```matlab
GridDiagChordsFlux = psitbxg2g(GridDiagChords,'Flux',psi)
GridDiagChordsFlux is a PsiTbx-Grid object =
  type: 'Flux-Coordinate'
  label: {1x3 cell  }
  storage: 'Time-Points'
  x: {1x3 cell  }
    t: [0.1000 0.3000 0.5000]
    par: [1x1 struct]
    csp: [1x1 struct]
plot(GridDiagChords.x{1},GridDiagChordsFlux.x{1}(:,1,:,:))
xlabel('S'), ylabel('Rho')
```
Function mapping on a Diagnostic–Chords coordinates

Mapping a function of the position on the chords of a diagnostic is a very essential operation in the Psi–Toolbox. The example shows the value of the canonical parabolic profile along the various chords of the diagnostic. Note that the value at the points along the chord and outside the definition domain of the function are set to NaN.

```plaintext
ParabolicRhoDiagChords = psitbxf2f(ParabolicRho,GridDiagChords)
ParabolicRhoDiagChords is a PsiTbx-Function object =
  x: [ 4-D  double ]
  grid: [81x2x20 psitbxgrid]
  t: [0.1000 0.3000 0.5000]
plot(ParabolicRhoDiagChords,'-')
```

or in another representation
<< |> Line integrals and averages

A function defined along Diagnostic−Chords can be integrated or averaged along the line with the `sum` and `mean` methods. In the example the line integral and average of the parabolic profile seen by each camera at different times is plotted as a function of the detector number.

```matlab
ParabolicRhodInt = sum(ParabolicRhoDiagChords,1)
ParabolicRhodInt is a PsiTbx-Function object =
  x: [2x20x3 double    ]
  grid: [1x1    psitbxgrid]
  t: [0.1000 0.3000 0.5000]
ParabolicRhodAvg = mean(ParabolicRhoDiagChords,1)
ParabolicRhodAvg is a PsiTbx-Function object =
  x: [2x20x3 double    ]
  grid: [1x1    psitbxgrid]
  t: [0.1000 0.3000 0.5000]
subplot(121), plot(permute(ParabolicRhodAvg.x,[2,1,3]))
subplot(122), plot(permute(ParabolicRhodInt.x,[2,1,3]))
```
The transfer matrix from a local emissivity defined on a cylindrical grid and chords can also be computed by the \texttt{intersect} method. The outer grid and a subgrid must be given to account for the shadow of the inner cylinder. In the following example an attractive plasma emitting the psi letter is seen by a 64 by 64 pixel CCD camera:

```matlab
t = linspace(0,60,64)/180*pi;
p = linspace(-30,30,64)'/180*pi;
tvd = repmat(t,length(p),1);
pvd = repmat(p,1,length(t));
dcd = psitbxdcd(1,0,0,pvd,tvd);
grid = psitbxgrid('c','g',
{linspace(.3,.9,21),linspace(-.6,.6,41),NaN});
T = intersect(dcd,[-Inf,Inf,-Inf,Inf],grid);
x = repmat('+',15:32);
x = x == '+';
x = reshape(T*x(:),size(dcd));
set(image(x),'cdatamapping','scaled'), axis equal off
```
intersect may also precalculate matrices that allow fast reconstruction of the emissivity profile.