ATMATCH function in AT

April 6, 2013

Abstract

This Document outlines the function *ATMATCH* using some examples.

Introduction

The *ATMATCH* function is a fitting routine implemented in the Accelerator Toolbox (AT) in order to reproduce the capabilities of the matching routines available for various accelerator physics codes. Given a certain number of variables and constraints, the selected minimization algorithm will try to fit the constraints by varying the variables. Among other features the routine is able to constraint any user defined function (for example transfer matrices between elements) and to define conditioned variables, keeping intrinsic conditions preserved. A vast flexibility of input is provided while taking care of the optimization of most frequent tasks. The capabilities of this routine are shown in this document using four main examples: the matching of linear optics parameters, the fit of a bump,matching of chromaticity and the fit of the lengths of two quadrupoles in order to make their gradients equal. This examples present all the aspects of interest of the present code.

Examples

Linear Optics Matching

The problem is to change the quadrupole gradients in order to achieve particular parameters in the lattice. The DBA cell structure used in the test is shown in figure 1, the parameters to be achieved are listed in table 1. The available parameters are the gradients of the four dipole families (QD,QF,QDM,QFM).

The code to obtain this matching is the following:

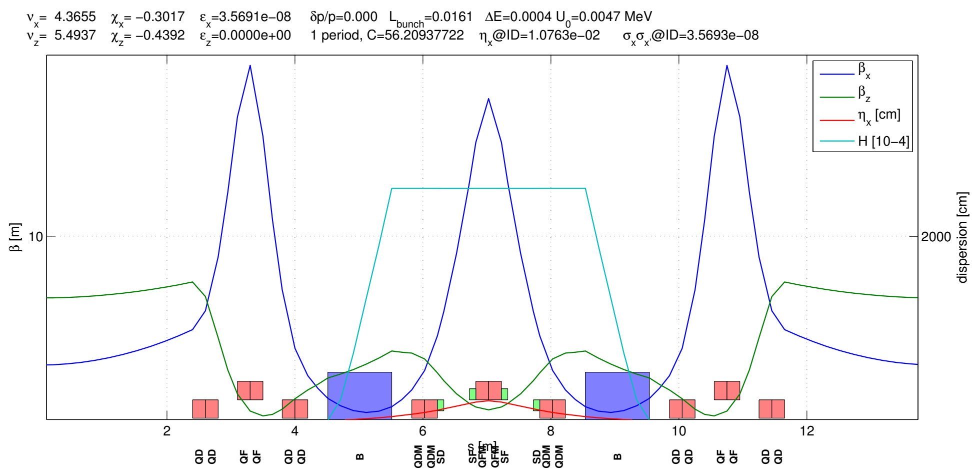


Figure 1: DBA ring cell for optics matching examples.

Table 1: Matching parameters for small DBA cell

	original	request	obtained
$\beta_x@QFM$	17.4838	17.3000	17.3000
$\eta_x@QFM$	0.0105	0	1.7758e-08
$\beta_y@QFM$	0.5514	0.5800	0.5800
μ_x@end	4.3655	4.3500	4.3500

```
% macro match dba test lattice beta functions and dispersion using
% quadrupoles.
%
% this macro shows the available basic functionalities of atmatch.
%
% various variable and constraint input constructions are shown
%
% the functions bety, betx, dispx, getDispersion are used by this example.

clear all
load('dba.mat','RING');
addpath(fullfile(pwd,'..'))

%%  VARIABLES
% Variab1=struct('Indx',{findcells(RING,'FamName','QD'),...
%                        findcells(RING,'FamName','QF')},...
%                 'LowLim',{[],[]},...
%                 'HighLim',{[],[]},...
%                 'Parameter',{{'PolynomB',{1,2}},{'PolynomB',{1,2}}}...
%                 );
% % % or alternative call
Variab1=atVariableBuilder(RING,{'QD','QF'},{{'PolynomB',{1,2}}});
```

```
% this variables use a function to change the gradeints of the QDM
% quadruoples
k1start=getcellstruct(RING,'PolynomB',findcells(RING,'FamName','QDM'),1,2);
Variab2=struct('Indx',{findcells(RING,'FamName','QFM'),...
                        @(RING,K1Val)VaryQuadFam(RING,K1Val,'QDM')...
                       },...
               'Parameter',{{'PolynomB',{1,2}},k1start(1)},...
               'LowLim',{[],[]},...
               'HighLim',{[],[]}...
               );

Variab=[Variab1,Variab2];

%%  CONSTRAINTS
qfmindx=findcells(RING,'FamName','QFM');
Constr1=struct('Fun',@(RING,~,~)dispx(RING,1),...
               'Min',0,...
               'Max',0,...
               'RefPoints',[],...
               'Weight',1);
disp('Horizontal dispersion at straigth section= 0')

% this call IS optimized. the function betx is using lindata at RefPoints
Constr2=struct('Fun',@(~,lindata,~)betx(lindata),...
               'Min',17.3,...
               'Max',17.3,...
               'RefPoints',[qfmindx(2)],...
               'Weight',1);
disp('Horizontal beta at QFM= 17.3')

% this call IS NOT optimized. the function bety is using twissring!
% the call to function mux is optimized, but requires all the reference
% points up to the last to be evaluated correctly
Constr3=struct('Fun',{@(RING,~,~)bety(RING,qfmindx(2)),...
                       @(~,ld,~)mux(ld)},...
               'Min',{0.58,4.35},...
               'Max',{0.58,4.35},...
               'RefPoints',{[],[1:length(RING)+1]},...
               'Weight',{1,1});
disp('Vertical beta at QFM= 0.58')
disp('Horizontal phase advance = 4.35')

Constr=[Constr1,Constr2,Constr3];

%%  CONSTRAINTS (Optimized using atlinconstraint)

% the following constraints reproduce the above definitions.
% the following constraints are optimized.
c1=atlinconstraint(qfmindx(2),...
    {{'beta',{1}},{'beta',{2}}},...
    [17.3,0.58],...
    [17.3,0.58],...
    [1 1]);

c2=atlinconstraint(1,...
    {{'Dispersion',{1}},{'tune',{1}}},...
    [0,0.35],...
    [0,0.35],...
    [1 1]);

c=[c1,c2];
```

```
%% MATCHING

% least square with non optimized constraints
RING_matched=atmatch(RING,Variab,Constr,10^-10,1000,3,@lsqnonlin);%

% fminsearch with non optimized constraints
RING_matched_optconstr=atmatch(RING,Variab,c,10^-4,1000,3);%
```

The function *ATMATCH* takes as arguments, the AT lattice, a variable structure and a constraint structure, the satisfactory value of the sum of squares of the constraints, the maximum number of iterations, the algorithm to use for the minimization and a verbose flag. The procedure is the following:

1 Get the current variable values

2 Apply a variable change

3 Get the constraints values

4 return constraints values to the matlab minimizer (return sum of squares to fminsearch)

5 loop until convergence criteria is reached.

The minimizers may be *fminsearch* or, if owned, *lsqnonlin*.

The *Variable* input may contain standard variables that change one of the fields of the AT lattice structure, or conditioned variables expressed as functions. The variables are expressed in a structure array with fields that describe which elements should be varied and which field in this elements is to be modified. The Variable structure fields may be:
If the variable is a property of an element in the lattice:

- *Parameter*: {Field to change, indxtomodify , ...} field in the AT lattice structure to be modified.
- *Indx*: indexes of the elements in the AT lattice structure to be varied
- *HighLim* and *LowLim*: variables range

If the variable is a function:

- *Parameter*: starting value for the variation
- *Indx*: @(ring,var)Fvariab(ring,var,...) any function with the signature that applies any change to the lattice.

- *HighLim* and *LowLim*: variables range

The capability to use functions as variables, allows as an example to perform the variation of multiple parameters in the lattice while keeping an extra constraint (condition). As an example it could be used to change the bending angles in the lattice while keeping the total bending angle constant (see later example).

The conditioned variables may also be empty. In this kind of declaration additional lattice elaborations, like a chromaticity rematch, are performed at every iteration (the *Parameter* field in this case is just and empty array).

In the Example above the function *VaryQuadFam* performs the same as the other variables but using a function. The function is reported here.

```
function R=VaryQuadFam(R,K1val,fam)
% functions returns a new ring with a different value for K1 of fam

indfam=findcells(R,'FamName',fam);
R=setcellstruct(R,'PolynomB',indfam,K1val*ones(size(indfam)),1,2);
```

To ease the definition of variables like quadrupole gradients, that are often present in common matching tasks, the function *atVariableBuilder* is provided. In the example, this function is used to define QD and QF gradients as two variables.

The *Constraints* are expressed in a structure array with fields specifying a function to be evaluated and the desired value.

The fields of the Constraint structure are:

- *Fun*: a function of the form: val=F(ATlattice,lineardata(RefPoints),globaldata,...). val has to be a row vector. The argument lineardata is the output of atlinopt while the argument globaldata is a structure with fields globaldata.fractune and globaldata.chromaticity.
- *Min*: minimum value of val (same size as val)
- *Max*: maximum value of val (same size as val)
- *RefPoints*: reference points vector
- *Weight*: weight to rescale various constraints (same size as val)

If the constraint functions may be defined using the *atlinopt* output, then the lineardata and globaldata field are used in order to minimize the calls to evaluate the constraints functions.

This Constraint structure instructs the code to call *atlinopt* only once for every distinct RefPoints position.

Anonymous functions like $@(\sim, lineardata, \sim)funct(lineardata)$ or $@(ATlat, \sim, globdat)funct(ATlat, par, indx, globdat)$ may be used to input functions that do not need all the arguments or require additional fixed arguments.

If the field RefPoints is left empty also the field lineardata will be empty.

In the example the functions bety, dispx and mux, are wrapper functions that ask AT to get the beta alpha and phase advance at a certain location (first kind constraint). For example the signature of betx is betxval=betx(arc,refpos).

Min, Max and Weight have the same dimension of the constraint function output (row vectors). The fields Min and Max are equal when matching to a given value. The penalty is evaluated as the distance from the closest of the two limits. The Weight parameter divides the value of the constraint, the Min parameter and the Max parameter.

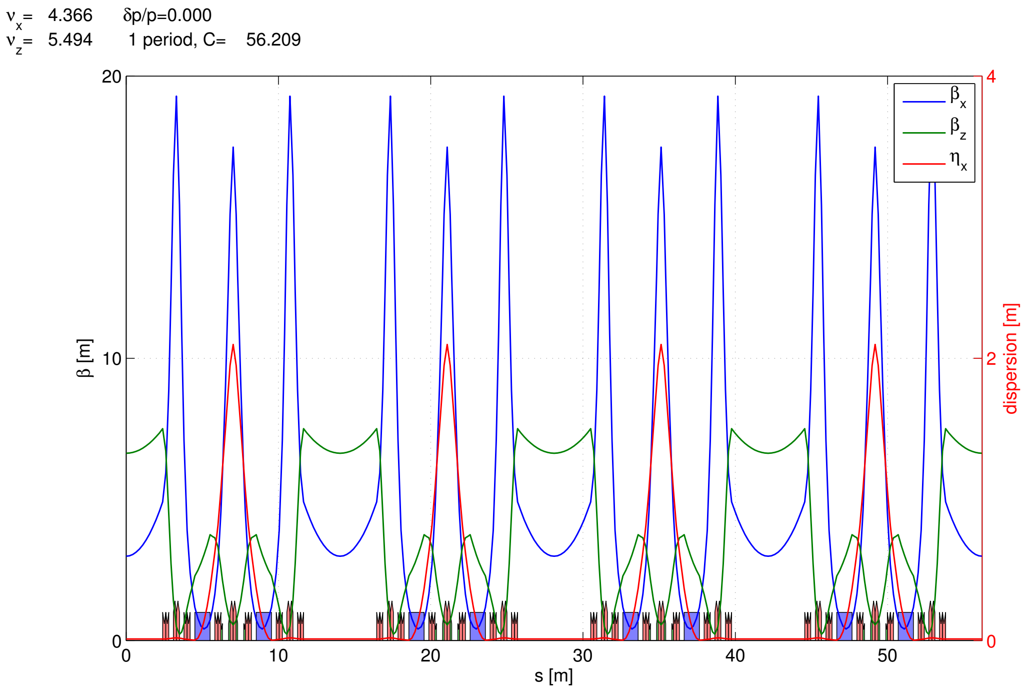
As an example to constrain the transfer matrix between two positions (for example a $-\mathcal{I}$ transformation between two sextupoles), it is sufficient to use as constraint a function like the one in the example following

```
function [r11]=RM44(lindata,ind1,ind2)
% get value of of indeces ind1 and ind2 (1 to 4) of
% M44 between two points first and last

Mlast=lindata(2).M44;
Mfirst=lindata(1).M44;

Mfirstinv=[[Mfirst(2,2),-Mfirst(1,2);-Mfirst(2,1),Mfirst(1,1)],...
            zeros(2,2);...
            zeros(2,2),...
            [Mfirst(4,4),-Mfirst(3,4);-Mfirst(4,3),Mfirst(3,3)]];

R=Mlast*Mfirstinv;
%R=Mlast/Mfirst;

r11=R(ind1,ind2);

end
```

The wrapping function in this example is necessary in order to provide the required output format for the constraint function (a row vector of scalars). The function is also exploiting the capability of *atmatch* to use in the functions definitions directly the lineardata structure output of *atlinopt* at the required locations.

The function *atlinconstraint* provides an easy definition of linear optics constraints. This function outputs directly structures that may be used in *atmatch*. In the example the same constraints are expressed using the structures and using *atlinconstraint*. The constraints generated by *atlinconstraint* are all taking advantage of the internal minimal call to *atlinopt*.

The result of the matching performed in the code above is shown in figure 2. The initial the required and the final parameters are listed in table 1.



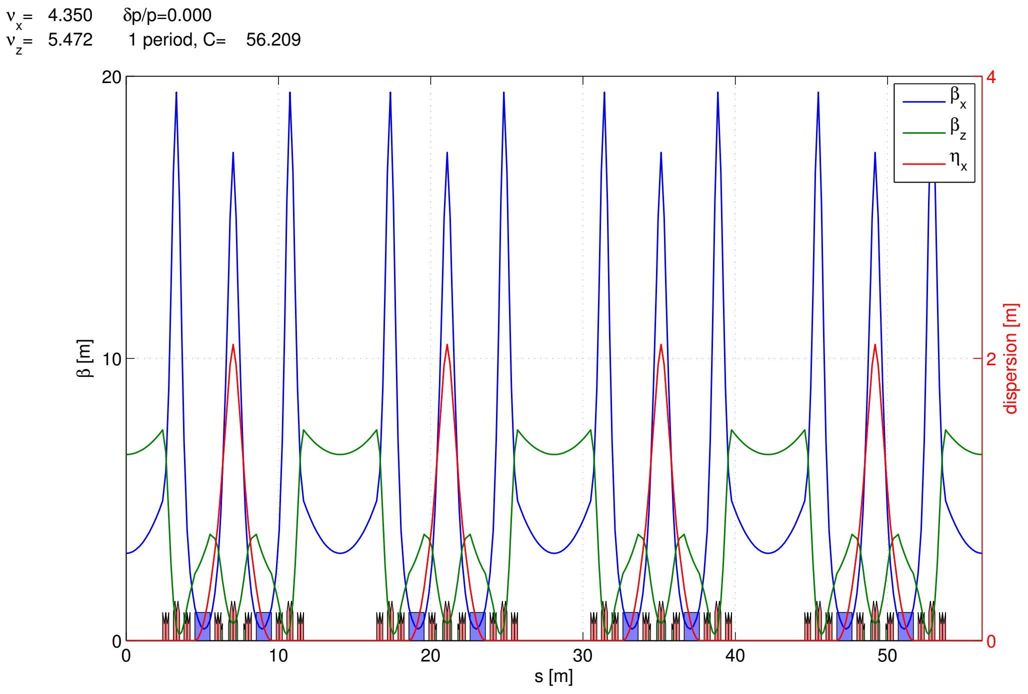


Figure 2: DBA ring before (top) and after optics matching.

The matching is performed using two different constraints definitions. The first definitions show how the constraint structure can be built in a versatile way. The second constraint definitions use the *atlinconstriant* function. This second call is limiting the coding necessary for standard optics fits and is using the intrinsic optimization provided in *atmatch* for constraints that may be evaluated using the *atlinopt* output structure.

Matching a Bump

Follows an example showing the matching of a bump at the straight section of the DBA lattice using 4 correctors.

```
% fit a bump using correctors
load('dba.mat','RING');

%correctors and BPM
C=atcorrector('C',0,0);
M=atmarker('BPM');

% get one cell and add elements
arc=[{M};RING(1:18);RING(128:end)];

%  corrector and BPM at every quadrupole center
indq=findcells(arc,'Class','Quadrupole');
for iq=2:2:length(indq)
    arc=[arc(1:indq(iq)-1);M;C;arc(indq(iq):end)];
    indq=findcells(arc,'Class','Quadrupole');
end

% build variables
hcor=findcells(arc,'FamName','C');

Variab=atVariableBuilder(arc,...
    {[hcor(1), hcor(end)],[hcor(2),hcor(end-1)]},...
    {{'KickAngle'}});

% build constraints
bpm=findcells(arc,'FamName','BPM');

c1=atlinconstraint(...
    [bpm(1)],...
    {{'ClosedOrbit',{1}},{'ClosedOrbit',{2}}},...
    [1e-3,0],...
    [1e-3,0],...
    [1e-2 1e-2]);

c2=atlinconstraint(...
    [bpm(2:end-1)],{{'ClosedOrbit',{1}}},0,0,1e-2); %#ok<*NBRAK>

c=[c1,c2];

% perform matching
arc_bump=atmatch(arc,Variab,c,10^-15,1000,3,@lsqnonlin);%'fminsearch',3);%
figure;atplot(arc_bump,@plClosedOrbit);
```

The resulting bump is shown in figure 3:

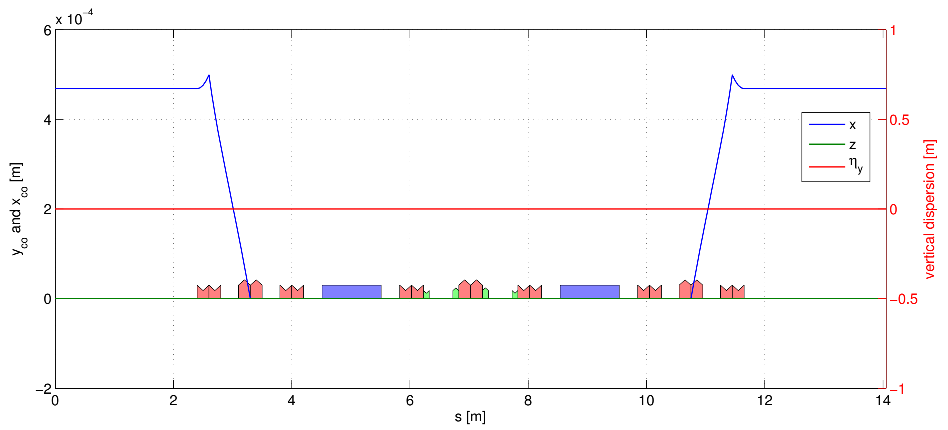


Figure 3: Bump in DBA cell.

In this example the variables are expressed using *atVariableBuilder* and the constraints are defined using *atlinconstraint*. Using this definition of the constraints *atlinopt* will be called only once and evaluated at all BPMs.

Matching chromaticity

A simple example to show how to perform a chromaticity matching using *atmatch*. The variable created use the chromaticity output of *atlinopt* as variable.

```
load dba.mat
VariabSxt=atVariableBuilder(RING,{'SF','SD'},{{'PolynomB',{1,3}}});

ConstrChrom=[atlinconstraint(1,{{'chromaticity',{1}}},0,0,1)...
             atlinconstraint(1,{{'chromaticity',{2}}},0,0,1)];

tol=1e-8;
RINGchrom0=atmatch(RING,VariabSxt,ConstrChrom,tol,1000,4);
```

Fit of the lengths of two quadrupoles to balance their gradients

The following fit routine varies the lengths of two quadrupoles in order to achieve equal gradients. The single variable used is the change in length of the quadrupoles, so the total length preservation condition is automatically fixed. This is expressed making use of a conditioned variable. The gradients of the quadrupoles are redetermined using the matching routine described in the first example defined as a conditioned variable without any value to vary. This Variable definition will perform a rematching of the optics (using the above function) at every iteration of the fit.

```
function arc1=MatchQuadLengthForEqualK1(arc,d)
% adjusts length of QF QFM to have the same K1
indQF=findcells(arc,'FamName','QF');
indQFM=findcells(arc,'FamName','QFM');
indQD=findcells(arc,'FamName','QD');
indQDM=findcells(arc,'FamName','QDM');

% initial matching
arc=matchingDBA(arc,17.3,0.58,0.0,0.35,indQF,indQD,indQFM,indQDM);

% variable definition
Variab1=atVariableBuilder(arc,{@(ar,dl)VaryMagLength(ar,dl,indQF,indQFM)},{0.01});

% empty varaible. Performs rematch at every iteration of the minimizer
Variab2=atVariableBuilder(arc,{@(r,~)matchingDBA(r,17.3,0.58,0.0,0.35,...
    indQF,indQD,indQFM,indQDM)},{[]});

Variab=[Variab1, Variab2];
%% constraints
Constr=struct( 'Fun',@(arc,~,~)compK1(arc,indQFM,indQF),...
            'Min',d,'Max',d,'RefPoints',[],'Weight',1); %

%% matching
tol=1e-6;
[arc1,~,currentvalues]=atmatch(arc,Variab,Constr,tol,100,3,@lsqnonlin);%'fminsearch',3);%

return

function RING_matched=matchingDBA(RING,bxDb,byDb,DDb,Qx,...
    indQF,indQD,indQFM,indQDM)

V=atVariableBuilder(RING,{indQF,indQD,indQDM,indQFM},{{'PolynomB',{1,2}}});

c1=atlinconstraint(indQFM(2),{{'beta',{1}},{'beta',{2}}},...
    [bxDb,byDb],[bxDb,byDb],[1 1]);

c2=atlinconstraint(1,{{'Dispersion',{1}},{'tune',{1}}},...
    [DDb,Qx],[DDb,Qx],[1 1]);

c=[c1,c2];

RING_matched=atmatch(RING,V,c,10^-6,1000,0,@lsqnonlin);%
return
```

Table 2: Fit of two quadrupoles length to obtain the same gradients

	original	request	obtained
LQFM	3.20000	-	3.41940
LQF	1.60000	-	1.38060
LQF+LQFM	4.80000	-	4.80000
$K1_{QF} - K1_{QFM}$	-1.02230	0	0

The functions used in the example to vary the magnets length and to retrieve the difference in gradient are reported below.

```
function r=VaryMagLength(r,DL,indP,indM)
% changes length of two mag families, keeping total length constant

ndp=length(indP);
ndm=length(indM);

r=setcellstruct(r,'Length',indP,getcellstruct(r,'Length',indP)+DL/ndp);
r=setcellstruct(r,'Length',indM,getcellstruct(r,'Length',indM)-DL/ndm);

return


function dif=compK1(arc,indQ1,indQ2,varargin)
% evaluates difference in gradient

KQ1=getcellstruct(arc,'PolynomB',indQ1,1,2);
KQ2=getcellstruct(arc,'PolynomB',indQ2,1,2);

dif=KQ2(1)-KQ1(1);

return
```

The result of the above fit (performed in 4 subsequent steps) are reported in table 2. The example shows the use of conditioned variables. The first variable is the function *VaryMagLength* that changes the length of the quadrupoles QF and QFM while keeping the total length constant. The second Variable is the matching of the lattice. This second variable has no input values, but performs a rematch at every iteration that changes the gradients of the quadrupoles. The constrained value (function *compK1*) is then the difference of the two gradients, that are varied by the matching routine. The choice to give the magnet positions as input to the matching routine function avoids multiple call to find the elements to vary in the lattice. This choice may improve the performance of the routine considerably.

Code

The code used is based on AT. The functions developed are:

- *ATMATCH* : minimizer. may minimize any function. used for optic matching and optimizations.
- *atlinconstraint* : creates constraints for linear optics. Any parameter output of atlinopt, may be used in the fit.
- *atVariableBuilder* : creates variables for simple parameters change in the AT lattice structure.

This functions and the above examples are now available in the pubtool directory of the AT code repository [3].

References

[1] MAD-X Home Page. `http://mad.web.cern.ch/mad/`.

[2] MAD8.13 user reference manual and Physics guide.

[3] Accelerator Toolbox `http://sourceforge.net/projects/atcollab/`

[4] Matlab `http://www.mathworks.fr/products/matlab/`

Other functions

```
function m=muy(lindata)

m=lindata(end).mu(2)/2/pi;

end

function [dx]=dispx(Seq,indx)
% get value of horizontal dispersion for  Seq(indx)

[dx,~]=getDispersion(Seq,indx);

end

function [dx,dy]=getDispersion(THERING,bpmindx)
% function [dx,dy]=getDispersion(THERING,bpmindx)
%
% determines dispersion taking two orbits at plus and minus DE=0.0001
%


DE=0.001;

Op=findorbit4(THERING,DE,bpmindx);
Opx=Op(1,:);
Opy=Op(3,:);

Om=findorbit4(THERING,-DE,bpmindx);
Omx=Om(1,:);
Omy=Om(3,:);

dx=(Opx-Omx)./(2*DE);
dy=(Opy-Omy)./(2*DE);

return

function [by]=bety(Seq,indx)
% get value of bety for  Seq(indx)

T=twissring(Seq,0,indx);
b=cat(1,T.beta);
by=b(:,2)';

end

function [bx]=betx(lindata)
% get value of betx for  Seq(indx)

b=cat(1,lindata.beta);
bx=b(:,1)';

end
```