IAIA Boundary Reconstruction Code,

Based on DOI 10.1016/j.fusengdes.2012.11.018.

Implemented by Rayan Sud, 11/09/2019, during internship
with ITER Organisation, supervised by Luca Zabeo.

```
function IAIA()
```

Set constants and data

Number of iterations:

```
N_ITERATIONS = 3;
```

Bounding points for X Point search grid

```
XP_RMIN = 4.7;
XP_RMAX = 5.4;
XP_ZMIN = -3.7;
XP_ZMAX = -3;
limiter = 0;
xpoint = 1;
```

Format of 'Geometry.mat': specify coordinates describing the inner ECs, the first wall, the guessed plasma shape, and the real plasma shape

Variables: *R_EC0* | *Z_EC0* | *R_wall* | *Z_wall* | *R_guess* | *Z_guess* | *R_realplasma* | *Z_realplasma* | *R_fullwall* | *Z_fullwall*

```
geometry = load('SampleData\Geometry.mat');
```

Format of 'MagneticData.csv': specify name, R-Z coordinates, angle, and

measured B field

Headers: *Name* | *R* | *Z* | *MainAxisPoloidalAngle* | *B*

```
magneticData = readtable('SampleData\MagneticData.csv');
```

Format of 'SourceCoils.csv': specify coordinates, current, and turns of active coils

Headers: *R* | *Z* | *I* | *n*

Subtract effect of active coils on magnetic data

```
sourceCoils = readtable('SampleData\SourceCoils.csv');
[sBr,sBz,~]= VectorizedGreenFunction(magneticData.R,magneticData.Z,sourceCoils.R,sourceCoils.Z
sourceB = sBr.*cosd(magneticData.MainAxisPoloidalAngle) + sBz.*sind(magneticData.MainAxisPoloid
sourceB = sourceB*(sourceCoils.I.*sourceCoils.n);
magneticData.B = magneticData.B - sourceB;
```

Get positions of inner ECs (equivalent currents) cleaning NaNs from CSV

```
R_EC0 = rmmissing(geometry.R_EC0);
Z_EC0 = rmmissing(geometry.Z_EC0);
```

Get positions of first wall cleaning NaNs from CSV

```
R_wall = rmmissing(geometry.R_wall);
Z_wall = rmmissing(geometry.Z_wall);
```

Get positions of initial plasma boundary guess cleaning NaNs from CSV

```
b0_R = rmmissing(geometry.R_guess);
b0_Z = rmmissing(geometry.Z_guess);
```

Set up outer layer of ECs, halfway between plasma guess and inner ECs

```
[R_EC1,Z_EC1]=iterateOuterECs(b0_R,b0_Z,R_EC0,Z_EC0);
```

Draw lines between inner ECs and first wall spacing = linspace(0,1,20);

```
lineZ = Z_EC0 + spacing.*(Z_wall-Z_EC0);
lineR = R_EC0 + spacing.*(R_wall-R_EC0);
```

Plot everything

```
fig=figure;
ax1 = subplot(1,2,1);
title(ax1,'Filaments & Boundary');
xlabel(ax1,'R (m)')
ylabel(ax1,'Z (m)')
axis(ax1,'equal')
xlim(ax1, [4 9])
ylim(ax1, [-5 5])

ax2 = subplot(1,2,2);
title(ax2,'Reconstructed Flux');
xlabel(ax2,'R (m)')
ylabel(ax2,'Z (m)')
axis(ax2,'equal')
xlim(ax2, [4 9])
ylim(ax2, [-5 5])

sgtitle('IAIA Reconstruction');
hold(ax1,'on')

realplasma = plot(ax1,rmmissing(geometry.R_realplasma),rmmissing(geometry.Z_realplasma),'m');
ec0plot=scatter(ax1,R_EC0,Z_EC0,'bx','HandleVisibility','off');
wallplot=plot(ax1,R_wall,Z_wall,'-ko','HandleVisibility','off');
linesplot=plot(ax1,lineR',lineZ','c','HandleVisibility','off');
```

```
boundaryplot=plot(b0_R,b0_Z,'--gd');
boundarypointplot=plot(ax1,b0_R,b0_Z,'--gd');
cPlot=scatter(0,0);
cPlot.Visible = 'off';
saddlePlot=scatter(0,0);
saddlePlot.Visible = 'off';
ec1plot=scatter(ax1,R_EC1,Z_EC1,'bx','HandleVisibility','off');
minPlot=scatter(ax1,-1,-1);
set(minPlot,'visible','off');
legend(ax1,'Real Plasma Boundary','Guess Plasma Boundary');
```

Iterative part, which runs for N_ITERATIONS loops.

```
for j=1:N_ITERATIONS
```

Solve for the currents in all the ECs, applying the inverse Green function matrix of B fields to the magnetic measurement data.

```
Ip0=solveForIp([R_EC0;R_EC1],[Z_EC0;Z_EC1],magneticData.R,magneticData.Z,magneticData.B,magnet
It=transpose([Ip0;(sourceCoils.I.*sourceCoils.n)]);
```

Find f;ux of outermost closed flux contour.

```
if limiter==1 && xpoint==0
```

If the configuration is set to a limiter, the method used is to find the mininum flux on the inner wall.

```
    delete(minPlot);
    [minFlux,minPlot]=findminFlux(ax1,[R_EC0;R_EC1; sourceCoils.R],[Z_EC0;Z_EC1; sourceCoils.Z
else
```

If the configuaration is set to x-point, the method used is to apply a gradient descent method to find a saddle point in the flux function, within the specified box coordinates.

```
    delete(saddlePlot);
    [minFlux,saddlePlot] = findXPointFlux(ax1,XP_RMIN,XP_RMAX,XP_ZMIN,XP_ZMAX,[R_EC0;R_EC1; sou
end
```

Find the new plasma boundary, with the points on the wall-to-outerEC lines with flux value closest to the found value from the x-point or limiter point. (finding the last closed flux contour)

```
delete(cPlot);
[b_R,b_Z,cPl]=calculateNewBoundary(ax2,R_EC0,Z_EC0,R_EC1,Z_EC1,R_wall,Z_wall,minFlux,sourceCoi
delete(boundarypointplot);
boundarypointplot=plot(ax1,b_R,b_Z,'--gd');
legend(ax1,'Real Plasma Boundary','Calculated Plasma Boundary');
cPlot = cPl;
```

Iterate the outer ECs to be midway between their old position, and the

new plasma boundary.

```
[R_EC1,Z_EC1]=iterateOuterECs(b_R,b_Z,R_EC1,Z_EC1);
delete(ec1plot);
ec1plot=scatter(ax1,R_EC1,Z_EC1,'bx','HandleVisibility','off');
end
```

Solves for the poloidal flux function on the inner wall, and returns the maximum value.

```
    function [minFlux,minPlot]=findminFlux(ax1,EcR,EcZ,It)
```

Solves for the poloidal flux function on the inner wall, and returns the maximum value.

```
leftLimiterWall = readtable('SampleData\LimiterLeftBoundaryWall.csv');
```

Repeat the matrix of all currents, to match the size of the Green matrix.

```
It_mat = repmat(transpose(It),[size(leftLimiterWall.R,1),1]);
```

Compute the Green matrix, for points on the wall

```
[~,~,Flux]=VectorizedGreenFunction(leftLimiterWall.R,leftLimiterWall.Z,EcR,EcZ);
```

Element-wise multiply the Green matrix with the repeated matrix of currents, and then sum the resultant matrix along the dimension of sources. This produces a vector of flux along the wall, due to all sources.

```
[minFlux,Idx]=max(sum(Flux.*It_mat,2));
minPlot=scatter(ax1,leftLimiterWall.R(Idx),leftLimiterWall.Z(Idx),'filled','r','HandleVisibilit
```

Returns a position midway between the boundary and outer ECs

```
    function [R,Z]=iterateOuterECs(b0_R,b0_Z,EC1_R,EC1_Z)
R=(b0_R+EC1_R)./2;
Z=(b0_Z+EC1_Z)./2;
```

Calculates the new plasma boundary

```
function [b_R,b_Z, cPlot]=calculateNewBoundary(ax2,EC0_R,EC0_Z,EC1_R,EC1_Z,wallR,wallZ,outerClo
```

Initialise the boundary arrays

```
b_R = zeros(size(wallR));
b_Z = zeros(size(wallZ));
```

Use MATLAB constrained nonlinear solver to minimise the difference between the flux and the desired contour flux, on each line (SLOW!!)

```
func = @(x)SquaredFluxError(x,It,outerClosedContourFlux,[EC0_R;EC1_R;sourceR],[EC0_Z;EC1_Z;sour
for i=1:size(EC1_R)
```

Here the vector 'x' has x(1) = R coordinate, x(2) = Z coordinate

Set the initial point to the midpoint of the line

```
x0 = [(wallR(i) + EC1_R(i))/2,(wallZ(i) + EC1_Z(i))/2];
```

The lower bound and upper bound are the end points of the lines.

```
lb = [min([EC1_R(i);wallR(i)]);min([EC1_Z(i);wallZ(i)])];
ub = [max([EC1_R(i);wallR(i)]);max([EC1_Z(i);wallZ(i)])];
```

The slope and constant are found to determine the equation of the line, y=mx + c

```
slope = (wallZ(i)-EC1_Z(i))/(wallR(i)-EC1_R(i));
c = wallZ(i) - slope*wallR(i);
 The linear equality constraint of the solver is set using the slopes and
 constants, reformulated into a matrix equation of the form Aeq * x = beq.
Aeq = [ slope, -1; slope, -1];
beq = [-c; -c];

minX = fmincon(func,x0,[],[],Aeq,beq,lb,ub);
b_R(i) = minX(1);
b_Z(i) = minX(2);
end
```

Produce contour plot from mesh, only for visualization purposes. Evaluate the flux function on a meshgrid, and plot.

```
x=linspace(4,9,50);
y=linspace(-5,5,50);
[X,Y]=meshgrid(x,y);
F = zeros(size(x,2),size(y,2));
for i=1:size(x,2)
    for j=1:size(y,2)
        [~,~,tF]=VectorizedGreenFunction(x(i),y(j),[EC0_R; EC1_R; sourceR],[EC0_Z; EC1_Z; sourc
        F(j,i)=sum(tF.*It,2);
    end
end

plasmaColorMap = load('plasmaColorMap.mat','plasmaColorMap');
colormap(ax2,plasmaColorMap.plasmaColorMap);
[~,cPlot]=contourf(ax2,X,Y,F,80,'HandleVisibility','off');
title(ax2,'Reconstructed Flux');
xlabel(ax2,'R (m)')
ylabel(ax2,'Z (m)')
axis(ax2,'equal')
```

Objective function for the minimisation of flux on lines

```
    function [error, gradient]=SquaredFluxError(x,I,outerClosedContourFlux,sourceR,sourceZ)
```

Provides the squared difference between flux of (x(1),x(2)) and the outerClosedContourFlux value, and the function gradient.

Compute the components of B and flux.

```
[Br,Bz,Flux]=VectorizedGreenFunction(x(1),x(2),sourceR,sourceZ);
```

Set the error.

```
error = (outerClosedContourFlux-sum(Flux.*I,2))^2;
```

Calculate the gradient, using known relations between components of B and flux function.

```
dFdr = x(1)*Bz.*I;
dFdz = x(2)*Br.*I;
derrordr = 2*(outerClosedContourFlux-sum(Flux.*I,2))*(-sum(dFdr,2));
derrordz = 2*(outerClosedContourFlux-sum(Flux.*I,2))*(-sum(dFdz,2));
gradient = [derrordr; derrordz];
```

Solve for the current distribution

```
    function Ip=solveForIp(EC_R,EC_Z,mCoilR,mCoilZ,mCoilB,mCoilAngle)
```

Solve for the Green function matrix, with sources as ECs and measured at measurement coil positions.

```
TRUNCATION_PERCENTAGE = 0.1;
[Br,Bz,~] = VectorizedGreenFunction(mCoilR,mCoilZ,EC_R,EC_Z);
```

This solves for the B field measured at each coil, using the angle of the coil.

```
Angle_mat = repmat(mCoilAngle,[1,size(EC_R)]);
B= Br.*cosd(Angle_mat) + Bz.*sind(Angle_mat);
```

Find the truncation index for a TSVD inversion, by finding the first value in the full SVD that is less than 1 of the maximum singular value

```
s=svd(B);
nnpct=find(s < TRUNCATION_PERCENTAGE*s(1));
k=nnpct(1);
```

Perform the TSVD inversion, and apply the inverse matrix to the magnetic measurements to obtain the currents.

```
[U,S,V]=svds(B,k);
Ip=sum(V.*repmat(transpose((transpose(U)*mCoilB)./diag(S)),size(V,1),1),2);
```

Use a gradient descent to find the x point.

```
    function [minFlux,saddlePlot]=findXPointFlux(ax,Rmin, Rmax, Zmin, Zmax, sourceR, sourceZ, s
```

Set the error tolerance in x-point position, and the step size of the gradient descent.

```
ERROR_TOLERANCE = 0.001;
STEP_SIZE = 0.05;
```

Set the initial point to the center of the box.

```
R0 = (Rmax + Rmin)/2;
Z0 = (Zmax + Zmin)/2;

scatter(ax,R0,Z0,'gx','HandleVisibility','off');
```

Set the initial error to be larger than the tolerance.

```
error = ERROR_TOLERANCE+1;

while error > ERROR_TOLERANCE
```

Find and plot the gradient.

```
[Br,Bz,~] = VectorizedGreenFunction(R0,Z0,sourceR,sourceZ);
dFdr = sum(R0*Bz.*sourceI',2);
dFdz = sum(R0*Br.*sourceI',2);
quiver(ax,R0,Z0,dFdr,dFdz,'HandleVisibility','off')
```

Find and apply the step in each coordinate

```
R_step = dFdr*STEP_SIZE;
Z_step = dFdz*STEP_SIZE;

R = R0 + R_step;
Z = Z0 + Z_step;
scatter(ax,R,Z,'rx','HandleVisibility','off')
```

Find the error as the Euclidean distance between the new and former position

```
error = sqrt((R-R0)^2 + (Z-Z0)^2);
R0 = R;
Z0 = Z;
end
```

Find and return the flux at the x-point position.

```
saddlePlot = scatter(ax,R0,Z0,30,'r','filled');
[~,~,minFlux] = VectorizedGreenFunction(R0,Z0,sourceR,sourceZ);
minFlux = sum(minFlux.*sourceI',2);
```