Orbit Code Approximations

Omar Leon

1 Introduction

The Orbit Code is a program which numerically calculates the trajectories of charged particles through magnetic field lines. The code required approximations to more accurately calculate detector efficiencies and particle trajectories. Two approximations that were considered are the geometry of the collimator and the total efficiency of the detectors as they are segmented into equal parts. Through more accurate calculations the Orbit Code can be used in conjunction with experimental data to arrive at models for the emissivity density of the fusion reactions in deuterium-deuterium plasmas.

2 Collimator Geometry

The Orbit Code has a square collimator which acts as the mechanical filter for the detector. The Charged Fusion Product Diagnostic utilizes a cylindrical collimator; as a result, it is necessary to calculate the deviation between the solid angle of acceptance between a square collimator and a cylindrical collimator.

Regardless of the shape of the collimator, the amount of particles that enter through the collimator and reach the bottom of it is dependent on the angle between the incoming particles and the collimator entrance, θ. Therefore, the amount of particles which enter the collimator is $A_{entrance}\cos\theta$ where $A_{entrance}$ is the area of the collimator entrance. The effective solid angle of acceptance of a collimator is given by equation 1, where T(θ) is the transmission coefficient which is determined by the shape of the collimator. Figure 1 depicts a cross sectional view of the relationship between the incoming particles and the collimator.

$$SA_{effective} = \int_0^{\theta_{max}} T(\theta) A_{entrance} \cos\theta \mathrm{d}\Omega \tag{1}$$

The transmission coefficient is the the area of overlap between the area created by the incoming particles extended to the vertical height of the detector and the active area of the detector. The area of overlap based on the type of collimator is shown in figure 2.

The results produced by equation 1 when using each collimator and the deviation between the two are shown in table 1.

3 Segmentation Efficiency

Using a square collimator, the Orbit Code is capable of segmenting the collimator entrance into equal parts along both sides of the square. This function allows for the

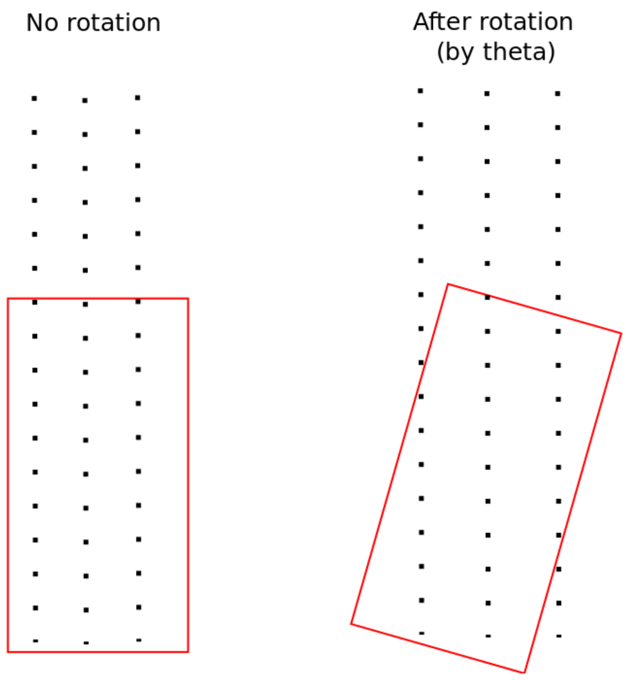
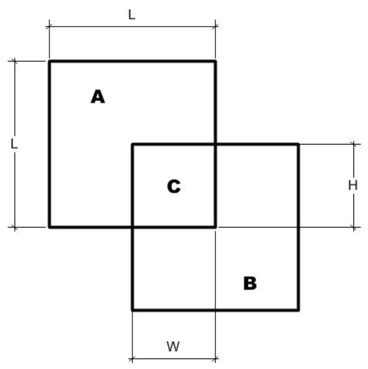
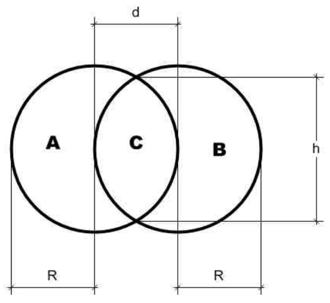


Figure 1: The red rectangle is a cross sectional view of the collimator and the black dots are incoming particles.



(a) Area of Overlap Created by a Square Collimator



(b) Area of Overlap Created by a Circular Collimator

Figure 2: Section A is the area of particles that have gone through the collimator entrance. Section B is the active area of the detector. Section C is the area of overlap between section A and B.

Table 1: Effective Solid Angle of Acceptance ($Sr\dot{m}^2$)

Square Collimator	Circular Collimator	Deviation (%)
9.83×10^{-8}	9.82×10^{-8}	0.1

determination of the amount of area a detector can probe; the more segmented the collimator entrance the better this scope is defined. In theory, the total efficiency of the detectors should be the same regardless of the number of segmentations were applied to the collimator entrance. However, the numerical calculations performed by Fortran have slight deviations in the total efficiency of the detector as the number of segments increases (figure 4).

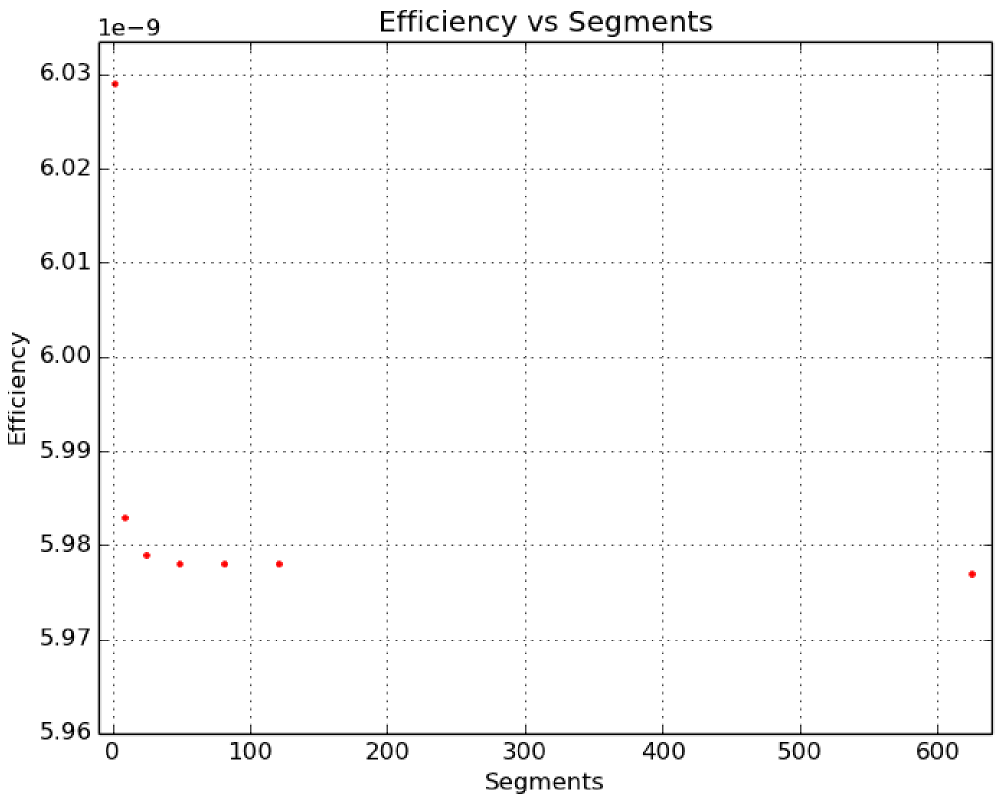


Figure 3: The total efficiency of the detectors notably decreases from 1 segment to 25 segments but stays at the same relative level as the segments increase beyond 25 segments.

Table 2: Total Efficiency vs Segments

Segments	Total Efficiency	Deviation of Total Efficiencies (%)
1	6.029×10^{-9}	0.
9	5.983×10^{-9}	0.76
25	5.979×10^{-9}	0.83
49	5.978×10^{-9}	0.85
81	5.978×10^{-9}	0.85
121	5.978×10^{-9}	0.85
625	5.977×10^{-9}	0.86

4 Conclusion

For now, the necessary approximations to the Orbit Code have been complete. The geometry for the collimator shape, such as converting from a square to a circular collimator, does not need to be changed in the Orbit Code. This is due to the deviation between the solid angle of acceptances to be less than 1%. In other words, the difference between the acceptances is small enough for us to accept the acceptance of a square collimator as a proper approximation. Additionally, no correction factor needs to be included when calculating the total efficiency of the detectors as the segmentation of the collimator increases. Even though there was a noticeable drop in efficiency between 1 segment and 9, the deviation between the efficiencies was less than 1%. Furthermore, the efficiencies seem to be asymptotic with respect to the number of collimator segments as the difference in efficiency between 121 segments and 625 segments is 0.001 (as shown in Table 2).

A Program to Calculate Solid Angle of Acceptance of a Cylinder

This follow is the code to calculate the solid angle of acceptance of a cylindrical collimator.

The Main_Accept program performs the integration of equation 1.

```
PROGRAM Main_Accept

use acceptancemod

implicit none

INTERFACE
       FUNCTION max_theta(r,d)
              REAL, INTENT(IN) :: r,d
              REAL :: max_theta
       END FUNCTION max_theta

       FUNCTION cc(theta, d)
              REAL, INTENT(IN) :: theta,d
              REAL :: cc
       END FUNCTION cc

       FUNCTION area_overlap(x1,y1,r1,x2,y2,r2)
              REAL, INTENT(IN) :: x1,y1,r1,x2,y2,r2
              REAL :: area_overlap
       END FUNCTION area_overlap

       FUNCTION acceptance(zeta)
              REAL, INTENT(IN) :: zeta
              REAL :: acceptance
       END FUNCTION acceptance
       FUNCTION acceptancearea(zeta)
              REAL, INTENT(IN) :: zeta
              REAL :: acceptancearea
       END FUNCTION acceptancearea

END INTERFACE

real :: s1,s2, angle_max


! Declare local variables

       angle_max = max_theta(rs,dm)
       call qsimp(acceptancearea, 0., angle_max, S1)
       call qsimp(acceptance, 0., angle_max, S2)
       write(*,*) '---------------------------------------------------'
       write(*,*) '---------------------------------------------------'
       write(*,*) '---------------------------------------------------'
       write(*,*) 'The acceptance is ', S1, 'steridians*m^2.'
       write(*,*) '---------------------------------------------------'
       write(*,*) '---------------------------------------------------'
       write(*,*) '---------------------------------------------------'
       write(*,*) 'The solid angle of acceptance is ', S2, 'steridians.'
END PROGRAM Main_Accept
```

The max_theta function calculates the angle at which no particle will enter the collimator entrance

```
!-----------------------------------------------------------------------------------------
!------------------------------------------------------------------------------------------
! Calculates the maxium angle of entry theta.

Real Function max_theta(r,d)

implicit none

real, intent(in) :: r,d

       max_theta = atan(2.*r/d)

end function max_theta
```

The cc function calculates the center of the "shadow" caused the particles entering the collimator entrance based on the angle between the incoming particles and the collimator.

```
!-------------------------------------------------------------------------------------------
!-------------------------------------------------------------------------------------------
! Calculates the center x point for the second circle.

real function cc(theta, d)

implicit none

real, intent(in) :: theta, d

       cc = d*sin(theta)/cos(theta)

end function cc
```

The area_overlap function calculates the area of overlap between two circles.

```
REAL FUNCTION area_overlap(x1,y1,r1,x2,y2,r2)

! This function is meant to calculate the area of overlap between two circles.
! Using the centers and radii of two circles their amount of overlap is
   calculated.

! Declaring the variables

use acceptancemod

implicit none
```

```
real, intent(in):: x1,y1,r1,x2,y2,r2

! Determining distace between the centers.

       write(*,*) 'The center of a circle is', [x1,y1],' with radius', r1,'.'
       write(*,*) 'The center of the other circle is', [x2,y2], 'with
          radius', r2,'.'

       distance = sqrt((x2-x1)**2 + (y2-y1)**2)
       write(*,*) 'Distance between the centers is', distance

! Determining if the circles overlap.

! If the distance between the centers is less than the sum of the radii the
   circles intersect or overlap.

       if (distance .lt. r1 + r2) then

! If the some of one radii and the distance between the centers is less than
   the other radii
! then the circles overlap.

              if (distance + r1 .le. r2 .or. distance + r2 .le. r1) then
                     ol = 0.

              else

                     ol = 1.


              end if

! If the distance between the centers is greater than the sum of the radii
   there is no overlap.

       else if (distance .ge. r1 + r2) then

              ol = 2.

       end if

! Calculating for the area of overlap if ol = 1.

       if (ol .eq. 1.) then

! Distance between x coordinates of centers.
```

```
            dx = abs(x2 - x1)
            write(*,*) 'The difference between the x coordinates of the
               center is', dx

! Distance between y coordinates of centers.

            dy = abs(y2-y1)
            write(*,*) 'The difference between the y coordinates of the
               center is', dy

! Center of the area of overlap.

            center_ellipse = abs((distance**2 + r1**2 - r2**2)/(2*distance))
            write(*,*) 'Center of the ellipse is', center_ellipse,'from the
               center of the circles.'

            intersectx(1) = x1 + dx*center_ellipse/distance +
               (dy/distance)*sqrt(r1**2 - center_ellipse**2)

            intersectx(2) = x1 + dx*center_ellipse/distance -
               (dy/distance)*sqrt(r1**2 - center_ellipse**2)


            intersecty(1) = y1 + dy*center_ellipse/distance -
               (dx/distance)*sqrt(r1**2 - center_ellipse**2)


            intersecty(2) = y1 + dy*center_ellipse/distance +
               (dx/distance)*sqrt(r1**2 - center_ellipse**2)

            write(*,*) 'The points of intersection are (',
               intersectx(1),',', intersecty(1),') and (',
               intersectx(2),',',intersecty(2),').'


! Calculating the area of intersection


            height = sqrt((intersectx(1) - intersectx(2))**2
               +(intersecty(2) - intersecty(1))**2)
            write(*,*) 'The height of the ellipse is', height

! First Circle of the Left

            if (x2 .ge. x1) then

            ! Geometric calculations.

                   theta = 2.*asin(height/(2*r2)) ! Angle of First Circle's
                      Sector
                   u1 = sqrt(r2**2 - (height/2)**2)
```

```
                area1 = theta*(r2**2)/2 - u1*height/2 ! Area of First
                   Circle


                if (distance > u1) then

                       phi = 2*asin(height/(2*r1)) ! Angle of Second
                          Circle's Sector
                       u2 = sqrt(r1**2 - (height/2)**2)

                       area2 = phi*(r1**2)/2 - u2*height/2 ! Area of
                          Second Circle


                else if (distance == u1) then

                       area2 = 0.5*pi*r1**2 ! Area of Second Circle


                else if (distance < u1) then

                       phi = 2*asin(height/(2*r1)) ! Angle of Second
                          Circle's Sector

                       u2 = sqrt(r1**2 - (height/2)**2)

                       area2 = (2*pi - phi)*(r1**2)/2 + u2*height/2 !
                          Area of Second Circle

                end if

                area_overlap = area1 + area2
                write(*,*) 'When OL=', ol,'the over lap area
                   is',area_overlap

!               First Circle on the Right
!
         else if (x2 .lt. x1) then


!               Geometric calculations.
!
                theta = 2.*asin(height/(2*r2)) ! Angle of Second
                   Circle's Sector
                u1 = sqrt(r2**2 - (height/2)**2)
                area1 = theta*(r2**2)/2 - u1*height/2 ! Area of Second
                   Circle

                write(*,*) 'Area1 is', area1

                if (distance > u1) then
```

```
                            phi = 2*asin(height/(2*r1)) ! Angle of Second
                                Circle's Sector
                            u2 = sqrt(r1**2 - (height/2)**2)

                            area2 = phi*(r1**2)/2 - u2*height/2 ! Area of
                                Second Circle
                            write(*,*) 'Area2 is', area2


                     else if (distance == u1) then

                            area2 = 0.5*pi*(r1**2) ! Area of Second Circle

                     else if (distance < u1) then

                            phi = 2*asin(height/(2*r1)) ! Angle of Second
                                Circle's Sector
                            u2 = sqrt(r1**2 - (height/2)**2)

                            area2 = (2*pi - phi)*(r1**2)/2 + u2*height/2 !
                                Area of Second Circle

                     end if

                     area_overlap = area1 + area2
                     write(*,*) 'When OL=', ol,'the over lap area
                         is',area_overlap


              end if

!             If there is a complete overlap.

       else if (ol .eq. 0.) then

              if (r1 < r2) then

                     area_overlap = pi*(r1**2)
                     write(*,*) 'When OL=', ol,'the over lap area
                         is',area_overlap


              else if (r1 >= r2) then
                     area_overlap = pi*(r2**2)
                     write(*,*) 'When OL=', ol,'the over lap area
                         is',area_overlap

              end if

!              If there is no overlap.
```

```
        else if (ol == 2.) then
                area_overlap = 0.
                write(*,*) 'When OL=', ol,'the over lap area is',area_overlap

        end if


end function area_overlap
```

The acceptancearea function provides equation 1 for the program to integrate.

```
!------------------------------------------------------------------------------------------
!------------------------------------------------------------------------------------------
! Calculates the acceptance with the direction of the particles taken into
   consideration

real function acceptancearea(zeta)

USE acceptancemod

implicit none


        Interface

        FUNCTION cc(theta, d)
                REAL, INTENT(IN) :: theta,d
                REAL :: cc
        END FUNCTION cc

        FUNCTION area_overlap(x1,y1,r1,x2,y2,r2)
                REAL, INTENT(IN) :: x1,y1,r1,x2,y2,r2
                REAL :: area_overlap
        END FUNCTION area_overlap

        End Interface

        real, intent(in) :: zeta
        real :: c2_area
        xp = cc(zeta, dm)
        ic = area_overlap(x,y,r,xp,yp,rp)

        if (xp == 0. .and. rp==r) then
                transmission = 1.
        else
                c1_area = pi*rp**2
                transmission = ic/c1_area
        end if

        c2_area = pi*rp**2
```

```
       acceptancearea = 2*pi*transmission*c2_area*cos(zeta)*sin(zeta)
end function acceptancearea
```

The acceptance function is a weighted integral of the solid angle of acceptance of a cylinder.

```
!-----------------------------------------------------------------------------------------------
!-----------------------------------------------------------------------------------------------
! Calculates the acceptance

real function acceptance(zeta)

USE acceptancemod

implicit none


       Interface

       FUNCTION cc(theta, d)
              REAL, INTENT(IN) :: theta,d
              REAL :: cc
       END FUNCTION cc

       FUNCTION area_overlap(x1,y1,r1,x2,y2,r2)
              REAL, INTENT(IN) :: x1,y1,r1,x2,y2,r2
              REAL :: area_overlap
       END FUNCTION area_overlap

       End Interface

       real, intent(in) :: zeta
       real :: c2_area
       xp = cc(zeta, dm)
       ic = area_overlap(x,y,r,xp,yp,rp)

       if (xp == 0. .and. rp==r) then
              transmission = 1.
       else
              c1_area = pi*rp**2
              transmission = ic/c1_area
       end if

       c2_area = pi*rp**2
       acceptance = 2*pi*transmission*sin(zeta)
end function acceptance
```

The makefile combines all of the files into an executable.

```
#-----Makefile for Soild Angle of Acceptance-------

%.o: %.f
```

```
	$(F90) -c $(FFLAGS1) $< -o $@

%.o: %.f90
	$(F90) -c $(FFLAGS) $< -o $@

Acceptance_OBJ = acceptancemod.o areaoverlap.o maxtheta.o cc.o qsimp.o
   trapzd.o MainAccept.o acceptance.o acceptancearea.f90

#INCLUDE = ../include/

F90    = gfortran

#FFLAGS = -ffixed-line-length-none $(FDFLAG) \
#         -fno-align-commons -w -fno-automatic -I$(INCLUDE)

FFLAGS = -ffree-form

FFLAGS1 =
#----------------------------------------------------------
all: $(Acceptance_OBJ)
#	if (test -f "acceptancemod.mod"); then rm acceptancemod.mod; fi
	$(F90) $(Acceptance_OBJ) -o accept

#example of creating object files the long way
#magfld.o: magfld.f
#	$(F90) -c magfld.f $(FFLAGS) -o $@
#----------------------------------------------------------

clean:
	rm *.o *.a *.mod accept testing

#acceptancemodcheck.mod: acceptancemod.f90
#	if (test -f "acceptancemod.mod"); then rm acceptancemod.mod; fi
#	$(F90) -c $(FFLAGS) $< -o /temp/xx

#acceptancemod.mod: acceptancemod.f90
#	$(F90) -c $(FFLAGS) $< -o acceptancemod.o

#area_overlap.o : area_overlap.f90
#	$(F90) -c $(FFLAGS) $< -o area_overlap.o

#max_theta.o : max_theta.f90
#	$(F90) -c $(FFLAGS) $< -o max_theta.o

#qsimp.o : qsimp.f
#	$(F90) -c $(FFLAGS1) $< -o qsimp.o

#trapzd.o : trapzd.f
#	$(F90) -c $(FFLAGS1) $< -o trapzd.o

#acceptance.o : acceptance.f90
```

```
#       $(F90) -c $(FFLAGS) $< -o acceptance.o

#Main_Accept.o : Main_Accept.f90
#       $(F90) acceptancemod.o area_overlap.o max_theta.o qsimp.o trapzd.o
   acceptance.o -c $(FFLAGS) $< -o Main_Accept.o
```