CHANDRA
X-ray Center 60 Garden St., Cambridge Massachusetts 02138 USA

MEMORANDUM

Date:	October 3, 2012
From:	Nancy RA. Wolk
To:	ACIS Operations Team
Subject:	Predicting the Amount of Earth in the ACIS cold radiator's field of view.
Cc:	Tom Aldcroft, Dan Schwartz
Version:	0.9

Abstract

This memo describes the methods used in the C++ code **earth_acis_FOV** which predicts the amount of Earth illuminating the *Chandra* ACIS cold radiator. In particular, the geometry of the elements in the code will be discussed and limitations that may impact the precision and accuracy of this model.

1 Why model the Earth illumination?

The ACIS focal plane (FP) temperature has been increasing over time during perigee passages. In 2009, occasional focal plane temperature excursions greater than 0.5°C during observations started to be observed and examined for the cause. One cause of this temperature increase was believed to be Earth illuminating the cold radiator. The ACIS cold radiator is strapped to the focal plane and radiates the heat from the focal plane out to space to keep the focal plane temperature stable at -120°C. However, during perigees, the apparent size of the Earth is large and the illumination of the Earth on the cold radiator can be enough to provide a heat flow to the focal plane. Behind the cold radiator, in the -Z direction, is the warm radiator, which is connected to the camera body to maintain a planned -60°C temperature. Protecting the radiators from space and the telescope are two irregularly shaped shades; the sun shade, which faces space and the telescope shade, which faces the SIM. The interiors of these shades are coated with goldized Kapton to reflect the radiated heat from the radiators out to space. Figure 1 displays the configuration of the radiators and shades.

To minimize the time when the Earth is in the field of view of the cold radiator, a Flight Operations Team Mission Planning (FOT MP) tool predicted the times when the Earth could be warming the focal plane via the cold

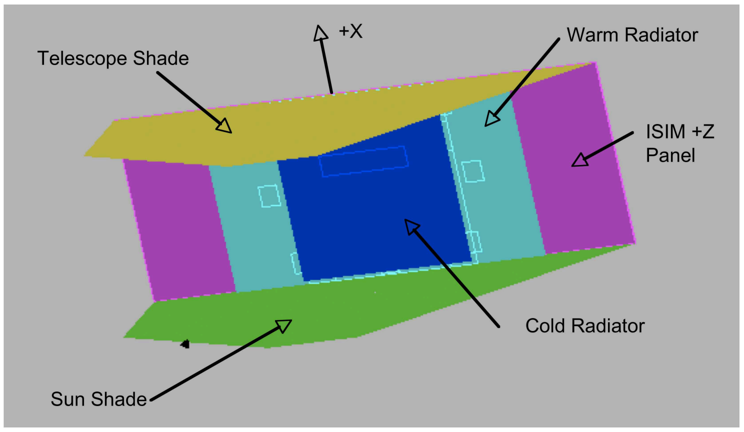

Figure 1: The ACIS cold and warm radiators with the sun and telescope shades. Support posts for the shades are not shown.

radiator. This tool only used a simple model of the shade shape and direct illumination from the center of the Earth. This memo discusses the conversion of this tool to C++ and the changes to the model. The FOT MP tool was converted to C++ and a Python prototype. The algorithm for the calculation of the direct and reflected illumination of the cold radiator was changed based on the Python code by Tom Aldcroft. All of the changes are discussed below. The goals of this work is to use this as an input to an ACIS focal plane temperature predictive model and to track the expected times of large Earth shine on the radiators.

2 Conversion of the FOT MP tool

The FOT MP Mission Constraint Checker (MCC) is a tool within MATLAB that allows mission planners to visualize different constraints while planning a command load. The angle of the Earth to the center of the cold radiators was one of the constraints. The inputs include the maneuvers for a week, and the Chandra centered Earth and Solar ephemerides. These inputs are stored in an internal structure within the MCC. An OFLS product, the maneuver summary file can be used as an input and is read and stored as this internal structure. The maneuver summary file simply contains the times of the quaternion update and the times of the maneuver start. A quaternion is a 4 dimensional representation of a rotation with the rotational angle combined with the three coordinate rotation axis to create the values. It is important to note that the quaternions are stored in an unconventional order with the values x,y,z,w instead of w,x,y,z. The code has been programmed to work with the former order. To convert a rotational axis (axis) and rotation angle (angle) to a quaternion, one must follow the steps in equation 1. The resulting quaternion must be normalized in case any value is close to zero.

$$\begin{array}{rcl} q.x &=& axis.x * sin(\text{angle}/2) \\ q.y &=& axis.y * sin(\text{angle}/2) \\ q.z &=& axis.z * sin(\text{angle}/2) \\ q.w &=& cos(\text{angle}/2) \end{array} \tag{1}$$

The MCC is written in MATLAB, and is driven by a graphical user interface. It interfaces with other MATLAB utilities such as the OR viewer and it uses the internal structure of maneuvers that is generated on the fly for planning a week. While this fulfills the needs of the FOT MP, it does not provide a good framework for the ACIS Ops load reviews.

With advice from the FOT, ACIS Ops converted the ACIS radiator section of the MCC to C++. In addition to the model changes, the code was redesigned as a stand alone C++ tool for use in the ACIS load reviews. One advantage of the MATLAB code is its object oriented nature. Classes were built to to match the MATLAB objects. Files could be read into classes that were then passed into various functions within the code. This makes integrating the changes back into MATLAB for FOT MP much easier. In addition, the classes used were mostly already built for another stand alone C++ tool, **bright_sources**, which determines when a bright X-ray source may be in the ACIS FOV. This tool was also based on an MCC code.

The classes migrated from **bright_sources** include *Maneuver*, which contains all of the code to create intermediate quaternions during slews and during holds, *Time*, a class to represent the time of an event based on different inputs, and quaternion, a class to perform specific quaternion mathematical operations. An *Ephemeris* class was added to manipulate the spacecraft, earth and solar positions. The *Time* class contains XTime objects from Arnold Rots' *XTime.C* code. This properly accounts for leap seconds and time zone conversions.

Once the MATLAB objects were matched with C++ classes, enhancements such as a more accurate representation of the radiator shades and cold radiator and reflections between the shades as contributors to the heating of the radiator were added to the code. Tom Aldcroft developed a new method to calculate the direct and reflected rays from the Earth striking the cold radiator. The original MATLAB algorithm was removed and replaced with this method. Figure 2 illustrates the differences between the MATLAB “taco” shape and the actual, asymmetrical, shape of the ACIS radiator shades.

The remainder of this memo will discuss the enhancements to the C++ code, and the limitations of this model. The appendices contain detailed documentation of the classes used.

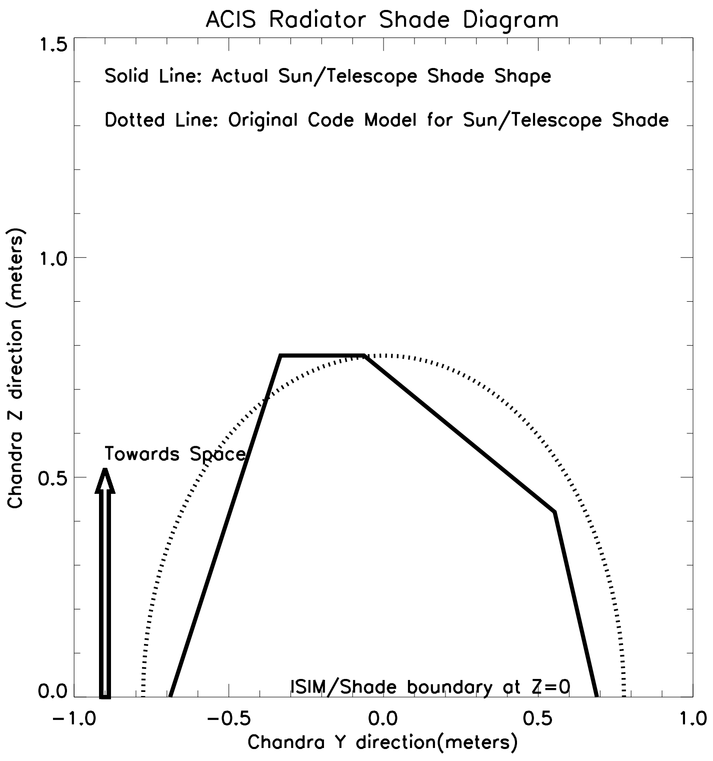

Figure 2: The dotted line displaying the original FOT model “taco” shape versus the actual shape (solid line) of the shades. While the “taco” is a good approximation, the actual asymmetrical shape changes the nature of the calculations.

2.1 Flow of code

The basic flow of the **earth_acis_FOV** code is stored in *main.cc* and described in Figure 3. The programming documentation of the *Maneuver*, *Time* and *Ephemeris* classes are in the appendices. The first step in this tool reads and interpolates the input files so the *Maneuver* and the *Ephemeris* objects are on the same time frame and interval. The seed times are the start and stop of the maneuvers for the week. This process synchronizes the ephmerides and maneuvers to allow for a sequential operation on the elements of the classes. The C++ standard template libraries Vector class is used to store the unit vectors, times and quaternions. Intermediate quaternions are calculated using *Chandra* parameters to replicate the movement of the spacecraft during maneuvers.

Each time sample, currently set to 60 seconds, runs through a loop that matches the correct quaternion with the corresponding Earth and Solar ephemerides unit vectors. The Earth positional vector is in Earth Centered Inertial (ECI) frame in kilometers. During the first interval, a random hemisphere of 1.5^6 positional vectors is created. This is the sample from which random positions on the Earth are collected for each time interval.

The size of the Earth grid is determined by the smallest possible unit x vector on the Earth as viewed by the cold radiator and forced to be between 100 and 10,000 points.

A corresponding number of random radiator points are generated. The actual calculation of the solid angle of the Earth in the cold radiator FOV is done in bf calc_earth_vis which determines if the rays from the radiator will intercept the positions on the Earth. The final solid angle is written to a text file in the main of the code.

2.2 Calculating the illumination of the radiators

The workhorse of **earth_acis_FOV** is the **calc_earth_vis** function. The flow of this function is illustrated in figure 4. This function takes the quaternion, the Earth positional vector, the altitude of Chandra from the center of the Earth,and calculates the solid angle of the Earth illumination that impacts the radiators at this time interval. The Earth illumination is broken down into rays that directly illuminate the cold radiator and rays that reflect off the radiator shade interior surfaces up to 10 reflections. After 10 reflections, the illumination has been attenuated enough to not have a significant impact on the heating of the radiator.

2.3 Creating a grid of points across the Earth

The calc_earth_vis function is executed once per time interval, however, this function loops over an array of rays between the Earth and the cold radiator. The first time calc_earth_vis is called, a static variable called “sphere_xyz” is created. This is a random hemisphere of points to represent the Earth’s surface. Since the most the cold radiator will see of the Earth is a hemisphere of the Earth, this hemisphere of unit vectors can be used multiple times as a pool of vectors to sample random parts of the Earth. The random hemisphere

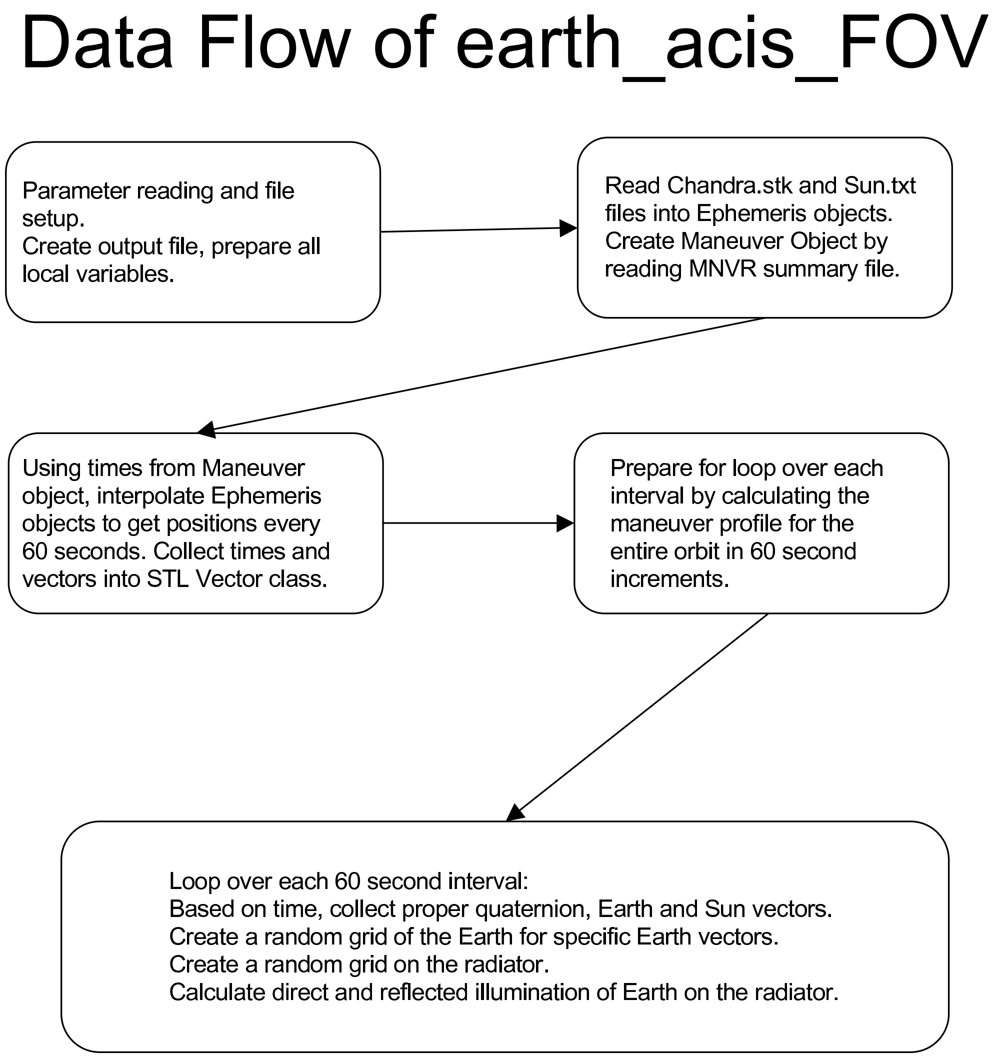
Data Flow of earth_acis_FOV
Parameter reading and file setup. Create output file, prepare all local variables.
Read Chandra.stk and Sun.txt files into Ephemeris objects. Create Maneuver Object by reading MNVR summary file.
Using times from Maneuver object, interpolate Ephemeris objects to get positions every 60 seconds. Collect times and vectors into STL Vector class.
Prepare for loop over each interval by calculating the maneuver profile for the entire orbit in 60 second increments.
Loop over each 60 second interval:
Based on time, collect proper quaternion, Earth and Sun vectors.
Create a random grid of the Earth for specific Earth vectors.
Create a random grid on the radiator.
Calculate direct and reflected illumination of Earth on the radiator.

Figure 3: The main flow of the **earth_acis_FOV** software.

has 1.5e6 unit vectors. A C++ STL Vector container is also returned that contains the X unit vectors to allow the selection from this pool.

Once the random hemisphere is created, at every 60 second interval, the minimum X unit vector can be determined by the altitude of the spacecraft and the radius of the Earth.

$$\begin{aligned} min_X &= cos(asin(R_\oplus/\text{altitude})) \\ gridsize &= min_P + (max_P - minP) * (1 - min_X) \end{aligned} \tag{2}$$

Where min_P is the smallest number of points possible in the random hemisphere grid and max_P is the size of the random hemisphere grid. The min_P is the higher of the ordinal position of the min_x in a the sorted random hemisphere or 100. The code will then randomly select $gridsize$ vectors from the random hemisphere to fill the positions on the Earth. This allows for both a large number of random samples to select from ($1.5e^6$) and a reasonable maximum sample size to calculate for each time interval($1.0e^5$).

These vectors will correspond to the apparent size of the Earth at this time interval and returns these unit vectors back to calc_earth_vis for creating rays between the cold radiator and the Earth. To create a particular set of rays from the cold radiator to the Earth, each point is rotated through the earth quaternion for that point in time. This ensures a different sampling of the Earth's surface each time interval.

2.4 Creating a grid of points across the cold radiator

The radiator grid must be calculated each 60 second interval to match the number of rays to the Earth. A loop over every grid element rotates the element through the Earth quaternion and creates a ray for each grid point center for this interval. If the resultant ray is positive in the Z direction and is greater than a minimum X value (to prevent small number issues), it is stored in a vector (rays_to_earth) to be passed to calc_earth_vis. Once all of the Earth grid elements have been determined, the cold radiator grid is built with the size of the rays_to_earth vector. The cold radiator generator creates two y positions (y and -y) for each x position to allow for a symmetrical representation around the y axis. The z position is fixed.

2.5 Does the radiator see the Earth?

To calculate the Earth illumination on the radiator, we use one of the radiator position points and one ray to the Earth and determine if they intercept. The Earth's ray could also reflect between the radiator shade interior surfaces and eventually illuminate the cold radiator. Each ray/radiator points pair is reflected up to 10 times. Each reflection is checked for the ray intercepting the cold radiator. The rays use an attenuation of 0.9 per reflection. This process is repeated for all of the points on the Earth grid that were determined to intercept the radiator. Once the ray is detected, the resultant illumination is added to a counter. After all rays to the Earth have been passed through the detection algorithm, the values in the direct and reflected counters are divided by the total solid angle to calculate the final illumination in this time bin.

3 Output of the earth_acis_FOV code.

The output of the **earth_acis_FOV** code is an ascii file that contains the time of the calculation, the the solid angle of the Earth for direct illuminations, the solid angle of the Earth for reflected rays and the pitch and off nominal roll of the spacecraft. The off nominal roll of the spacecraft does not always match what is available in MATLAB and is of questionable value. The pitch is correct.

4 Limitations of the model.

As with all software and models, there are limitations in this code. These limitations may not give a completely precise measure of the solid angle of the earth, but the model does a mostly accurate job.

The main limitations are:

- Predicted vs Actual ephemerides.
- Reflectivity of the Interior Surfaces.

The first limitation is the use of the predicted ephemerides. The software uses the predicted Solar and Chandra ephemerides from the FOT. These data are calculated using JPL software. The Chandra ephemeris is created in an stk file, usually in a 200-300 day period. Using MATLAB, the Solar ephemeris can be calculated. Since this is predicted, the values may be off, but most likely, these are only small variations. A future enhancement is for the software to use the ephemerides that are released by the FOT once a week. A difficulty is to determine how to collect the solar ephemerides.

The reflections also use a reflectivity of 0.9, although the real value may be different. The number suggested by thermal documents is 0.98, but this value was lowered to account for possible changes to the surfaces over the mission. The code also makes the assumption that the entire illumination is reflected in toward the radiator. This is not true, but it works for an approximation.

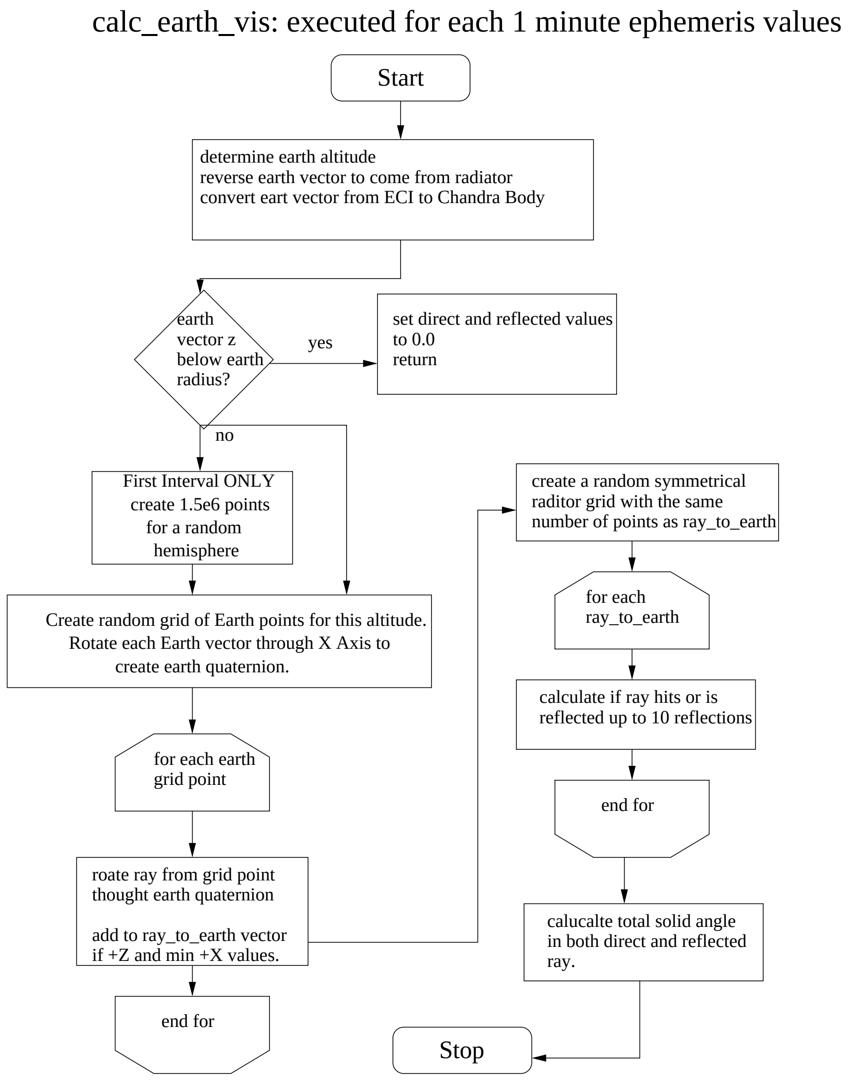
calc_earth_vis: executed for each 1 minute ephemeris values
Start
determine earth altitude
reverse earth vector to come from radiator
convert eart vector from ECI to Chandra Body
earth vector z below earth radius?
yes
set direct and reflected values to 0.0
return
no
First Interval ONLY create 1.5e6 points for a random hemisphere
Create random grid of Earth points for this altitude.
Rotate each Earth vector through X Axis to create earth quaternion.
for each earth grid point
roate ray from grid point thought earth quaternion
add to ray_to_earth vector if +Z and min +X values.
end for
create a random symmetrical raditor grid with the same number of points as ray_to_earth
for each ray_to_earth
calculate if ray hits or is reflected up to 10 reflections
end for
calucalte total solid angle in both direct and reflected ray.
Stop

Figure 4: Flow of the code that calculates the illumination of the radiators.

The code was tested against three tools: the MCC code to confirm the center angles matched, the Spacecraft Tool Kit (STK) and Tom Aldcroft's python prototype. The STK software visualizes *Chandra* and the relative positions of the Earth and Sun. A projection of the Earth's illumination was added and this confirmed when the **earth_acis_FOV** code predicted illumination of the radiator. See the memo TBD in edit phase for a detailed analysis of the comparison between the Python and C++ software.

5 Future Enhancements

The ultimate goal of this software is to allow the prediction of the ACIS FP temperature. Since this tool was written, the main has been pulled into the Python temperature modeling framework, Xija. This allows for the used of the Earth heating on the cold radiator to be used for the ACIS FP model. As of September, 2012, the ACIS FP model, which uses the C++ code here, allows the FOT MP staff to determine the best thermal situations for a week during the planning.

Expected enhancements may include calculating the heating power on the radiators rather than the solid angle of the Earth. This would allow the conductive transfer from the shades to be included. Another enhancement may to be returning the expected heat flow towards or away from the focal plane in a closed "ACIS FP-Cold Radiator" system. While this system is not quite physical, it is another step closer to the actual temperature predictions. Another enhancement will be to also determine the Earth on the Warm Radiators to predict the heat transfer to the ACIS camera body.

Appendix

This code used object oriented design in C++. The following appendices describe the classes used, the variables included and the methods. This will serve as documentation for the code in addition to the comments within the code source.

Appendix A Ephemeris Class

This class reads, stores and interpolates the ephemeris for the sun and Chandra. The class contains the following variables:

```
private:
  vector<Vector3D> center;       // The x,y,z position
  vector<Vector3D> velocity;     // the velocity in the 3 space
  vector<double> length;         // the length of each 3 space vector
  vector<Vector3D> unit;         // The unit vector of the center
  string target;                 // comparison target
  vector<Time> time;             // Time array
  bool apparent;        // 0 for real position velocity, 1 for apparent pos/vel
  int rows;                  // rows read into the arrays from eph files
  double radius;  // target's radius. Set to 1 as default.
```

Most of the variables are stored in a Standard C++ Library vector class. This is just to set up a FIFO for the code to loop over. The class Vector3D will be discussed later. The variables in the class are documented as to the meaning. One ephemeris file is stored in a single Ephemeris object.

A.1 Public Methods

The publicly accessible methods are:

```
public:
  Ephemeris(); // Constructor
  ~Ephemeris(); // Destructor
  Ephemeris(const Ephemeris&);//Copy Constructor
  bool operator=(const Ephemeris&);//Assignment
  //ACCESSORS
  void setCenter(vector<Vector3D>); // set Center position
  void setVelocity(vector<Vector3D>); // set center velocity
  void setTarget(string);         // set Target name
  void setTime(vector<Time>); // set Time of position
  void setApparent(bool);         // set the apparent information
  vector<Vector3D> getPosition();       // return the stored position
  vector<Vector3D> getVelocity();       // return the stored velocity
  vector<double> getLength();           //return the stored Length
  vector<Vector3D> getUnit();           //return the stored unit vector
  vector<Time> getTime(); // return the stored times
  string getTarget();                   //return the targetname
  double getRadius();         // return the radius of the object
  int getRows();         // return the size of the file

  //Printing and actions on the Ephemeris object
  void print();    // print the items in the object
  int  readEphemeris(string ,string); // read the emphermis file
  int Interpolate(Time , Time, int, Ephemeris &); //cubic spline
```

Most of the public members are accessor functions to read or set a variable in the class. The set functions allow for a manual override of a file for calculation testing on a small sample of positions. The two important public member functions are readEphemeris, which reads in the files and populates the class and Interpolate. Interpolate uses the input start and stop times and a step size in seconds. Then the ephemeris is interpolated using a cubic spline to rebin the data to start at the start time over the step sizes. This allows for the Maneuver (see below) and the Ephemeris objects to have the same start times and time resolution.

A.2 Dependencies

The Ephemeris class requires the following header files from the C++ distribution:

```
<iostream>
<cctype>
<cstdlib>
<string>
<vector>
<cassert>
<fstream>
<iomanip>
```

The local classes used within Ephemeris.cc are: "Time.hh" and "Vector3D.hh" and the code declares that it uses the std namespace. Additionally, the C++ class XTime is now used to track the timing to prevent local host machine timing issues.

Appendix B Vector3D Class

This simple class stores a three dimensional vector, just to keep it easier to pass these around the functions and classes.

```
private:
  double x;
  double y;
  double z;
```

B.1 Public Members

The public member functions are defined below:

```
public:
  //Constructors
  Vector3D();
  Vector3D(double[]);
  Vector3D(double,double,double);

  //destructor
  ~Vector3D();

  //Copy Constructor
  Vector3D(const Vector3D&);

  //Assignment operator
  bool operator=(const Vector3D&);

  //Accessors
  double getX();
  double getY();
  double getZ();
  void setX(double);
  void setY(double);
  void setZ(double);

  //Functions
  double getLength();
  Vector3D getUnitVector();
  void print();
  void print(std::ostream&);
```

The print function is an overload of the << operator. This allows for easier printing and debugging of this class.

B.2 Dependencies

The following standard C++ headers are required. This file must be linked with the math libraries at compilation time.

```
<cstdio>
<cctype>
<cstdlib>
<string>
<cassert>
<iostream>
<cmath>
```

Appendix C Maneuver Class

The Maneuver class tracks the maneuvers of the spacecraft. This code involves the majority of the calculations as it must follow a detailed pathway to track the intermediate quaternions during a slew. It is important to remember that in a slew, there are various jerks and accelerations as the telescope starts and stops the slew. The calculations for the intermediate quaternions are discussed below.

```
private:
  vector<Quaternion> manProfile; // Profile for this maneuver
  vector<Time> manTime;        // Time vector for maneuver profile
  Quaternion initial_Q; // the initial Quaternion for this maneuver
  Quaternion final_Q; // the final Quaternion for this maneuver
  Time start_time; // the start time of this maneuver
  Time stop_time;               // the stop time of this maneuver
  double duration; // time for this maneuver in seconds
  double * sarray; // pointer to temp array
  // intermediate products polar coordinates needed to calculate maneuvers
  double * phi;
  double * dphi;
  double * d2phi;
```

C.1 Public member functions

```
public:
  //CONSTRUCTOR FUNCTIONS
  Maneuver(Quaternion, Quaternion, double); //CONSTRUCTOR
  ~Maneuver();             // Destructor
  Maneuver(const Maneuver & );              // constructor
  Maneuver operator=(const Maneuver&);      //copy constructor

  //main workhorse
  void makeTrends(int);                     // get intermediate Quats
  double manvrTime(Quaternion,Quaternion);  // find time between 2quaternions

  //ACCESSORS
  vector<Quaternion> getManProfile();       // return intermediate Quat
  vector<Time> getManTime();                //return intermediate Time
  Time getStart();                          //return start of maneuver
  Time getStop();                           //return stop of maneuver
  Quaternion getFinal();                    //return final quat
  Quaternion getInitial();                  //return init quat
  void print();                   //Print the object
  void printInfo();                         //print info in object
```

The two functions of interest are makeTrends and manvrTime, are based on MATLAB code and use the characteristics of Chandra to calculate the time it takes to maneuver the telescope and the actual jerk and acceleration of the telescope as it slews. The code was completely converted from the MATLAB code. It breaks the slew into 7 segments and determines, using polar coordinates, the quaternion for the time interval specified. This quaternion is returned to allow the code to know exactly where the telescope is pointing at any time during an orbit.

C.2 Dependencies

This class requires the following header files. Three macros are defined to match the Chandra slew characteristics.:

```
iostream
```

```
cctype
cstdlib
string
vector
cassert
"quaternion.hh"
"Time.hh"

//CHANDRA HARDWARE CHARACTERISTICS
#define ALPHAMAX 2.18166e-6
#define DELTA 60.0
#define VMAX 0.001309
```

Appendix D The quaternion class

The pointing of the spacecraft is specified in a 4 dimensional vector called a quaternion. Quaternions contain the x,y,z positions and w, the rotation. This class was originally written by Angela Bennett as free ware. It was modified to work with the current C++ and to only use double type quaternions as opposed to the original container class.

```
private:
  double w, x, y, z;
```

D.1 Member Functions

```
public:

  //constructors
  Quaternion(void);
  Quaternion(double wi, double xi, double yi, double zi);
  Quaternion(double v[4]);
  Quaternion(const Quaternion& q);
#ifdef SHOEMAKE
  //Quaternion
  // -parameters : yaw, pitch, and roll of an Euler angle
  // -creates a new quaternion based on the Euler elements passed in
  // -used with Shoemakes code
  Quaternion(double e[3], int order);
#endif
```

```
   // -default destructor
   ~Quaternion();

   //Overloaded operators
   Quaternion operator = (const Quaternion& q);
   Quaternion operator + (const Quaternion& q);
   Quaternion operator - (const Quaternion& q);
   Quaternion operator * (const Quaternion& q);
   Quaternion operator / (Quaternion& q);
   Quaternion& operator += (const Quaternion& q);
   Quaternion& operator -= (const Quaternion& q);
   Quaternion& operator *= (const Quaternion& q);
   Quaternion& operator /= (Quaternion& q);
   friend inline ostream& operator << (ostream& output, const Quaternion& q)
   friend inline ostream& operator << (ostream& output, const Quaternion* q)
   bool operator != (const Quaternion& q);
   bool operator == (const Quaternion& q);

   //ACCESSORS
   double getZ();
   double getY();
   double getX();
   double getW();
   double setW(double wi);

  //Quaternion Math
   Quaternion pos_mult (const Quaternion& q);
   double norm();
   double magnitude();
   Quaternion scale(double s);
   Quaternion inverse();
   Quaternion conjugate();
   Quaternion UnitQuaternion();
   void UnitVector(double v[3]);
   void QuatRotation(double v[3]);
   double Quat2unitRotation(double v[3]);
   void Quat2RADec(double v[3]);

   //EULER CODE

#ifdef SHOEMAKE
```

```
  // - converts this quaternion into Euler angles
  void toEuler(double e[3], int order);
#endif
```

The earth_acis_FOV code does not use the Euler angles, but it is documented for completeness.

D.2 Dependencies

```
iostream
math.h
#ifdef SHOEMAKE
#include "EulerAngles.h"
#endif
#define PI 3.1415926535898
```

The EulerAngles.h file will not be discussed here. Please refer to the code base for more information.

Appendix E The Time Class

The Time class was written to convert the different forms of time input strings and store them as seconds in XTime. This allows for easier interpolation, subtractions and conversions. XTime.C is a C++ class written by Arnold Rots to support time conversions between different time systems. This fully accounts for leap seconds and prevents a bug that was caused by using the time system on a local machine.

```
private:

  XTime item_time;                           // the item time
  bool readString(std::string);              //parse a string
  bool readItems(int,int,int,int,int,double);//read the individual items
  bool readManvTime(double);                 //read time in MNVR file
  void convertDOY(int,int);                  //convert the DOY
```

E.1 Public Member Functions

```
public:
  //Constructors
  Time(); // constructor
  Time(std::string); // constructor with a colon delimited string
  Time(int, int,int,int,int,double); // constructor with Year,month,day,min,
                  // hour,seconds
  Time(double);   // using MNVR file format
  Time(time_t);   // using a time_t item
  Time(const Time &);             //Copy Constructor
  ~Time();   // destructor

  //operator overloads
  bool operator= (std::string);   //Assignment operator/ variable STRING
  bool operator= (double);        //assignment operator/  variable DOUBLE
  bool operator= (time_t);        //assignment operator/   variable time_t
  Time& operator= (const Time &); //assignment operator/  variable Time
  Time operator+ (Time);          //addition operator
  Time operator+ (int);   // addition overload to allow seconds to
                                  //be added
  double operator- (Time);        //subtraction operator
  bool operator> (Time);
  bool operator< (Time);
  bool operator== (Time);

  //printing and accessors
  void print();                   //print
  void printTime(std::ostream& ); //printTime
  std::string getString();        // return date string
  time_t getTime();   // return time struct
  double getSeconds();            //return time in seconds
  void setPtr();                  //resetPTr based on time_t

//overload the << operator as a friend
friend
inline std::ostream & operator<<(std::ostream& out,
     Time &obj)
```

The public members of the Time class are mostly overloads to allow for easier manipulation of the event times. The different files read by the other classes have different formats, so the idea is to create a fixed internal format.

E.2 Dependencies

The following libraries are required for the Time class.

```
cstdio
cctype
cstdlib
string
sstream
cassert
ctime
Xtime.h
iostream
```