propcov-cpp description

# Version history

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Notes** |
| 9 December 2021 | 0.1 | First draft, adapted from Orbit & Coverage design doc 1890325 v1.2.doc |
|  |  |  |
|  |  |  |
|  |  |  |

# Overview

This document describes the C++ classes of the propcov C++ library (in the lib/propcov-cpp/ folder). Most of the classes have been developed through the TAT-C project. The primary utility of these classes is to provide for spacecraft orbit propagation and coverage calculation.

Modifications have been made to original TAT-C classes, and additionally classes corresponding to the following two functionalities have been added:

1. A new point in spherical polygon algorithm.
2. Projection of sensor detector arrays to ground pixels.

The document comprises two main sections, the first being a description of the interfaces and a high level description of system behavior. The second main section documents the system structure in more detail, defining class responsibilities, class dependencies, key data structures and key functions. The system structure in turn is divided into two sections, one covering the Propagator and Spacecraft, the other the CoverageChecker.

In addition to the descriptive documentation, Doxygen output for all the source code is included as an appendix.

# Interface Description

This section provides information needed to implement the propcov-cpp C++ classes for spacecraft orbit propagation and coverage calculations. (Note that in OrbitPy the classes are wrapped with PyBind11 and used in a python setting.) It provides the interface to key routines used to access propcov-cpp capabilities, including precise definitions for each argument.

It also provides high level descriptions of the propcov-cpp subsystem’s behavior. This is intended to provide a broad outline; the details are provided in the source code itself and in the System Structure section of this document.

## Calling Key Routines

### Propagate

The Propagate function is defined in the Propagator class, and has the following signature:

virtual Rvector6 Propagate(const AbsoluteDate &toDate);

Argument **&toDate** – this is an AbsoluteDate object; class AbsoluteDate provides the ability to represent dates as either Julian or Gregorian dates. Generally Gregorian dates are used for initialization and Julian dates used for computations. The propagator will propagate the spacecraft’s state to that time.

Return value **Rvector6** – this is a 6 element vector of real numbers representing the spacecraft state. The first three elements of this vector represent the spacecraft’s position in Earth-centered inertial coordinates, the next 3 represent the velocity in the same coordinate frames. These two vectors are in kilometers and km/sec, respectively.

### CheckPointCoverage

The CheckPointCoverage function is defined in the CoverageChecker class and is used to determine the presence/absence of a ground-point (lat,lon coords) within a sensor/spacecraft Field-Of-View(FOV). There are several overloaded functions, and the below one is expected to be of typical use.

virtual IntegerArray CheckPointCoverage();

It returns array of integers contains indices of points from a PointGroup (see discussion in System Structure section of this document) that are visible at a given time (provided by the Spacecraft object referenced in the CoverageChecker class).

## High Level Behavior

This section gives a high-level view of how pieces of the general Initialize-Propagate-Coverage use case work. They are presented as descriptive text and snippets of actual code that show the key concepts of how this subsystem is intended to be used. More details are in the “System Structure” section of this document.

### Initialization

In the system test driver, the classes initialize in the following order. Dependencies on predecessor classes are listed for each class.

* LagrangeInterpolator – none
* Earth – none
* AbsoluteDate – none
* OrbitState – none
* Sensor subclasses (ConicalSensor, ~~RectangularSensor~~, GMATCustomSensor, DSPIPCustomSensor) – none
* NadirPointingAttitude – none (a subclass of Attitude)
* Spacecraft (Attitude, AbsoluteDate, OrbitState ,LaGrangeInterpolator)
* Propagator (Spacecraft)
* PointGroup
* CoverageChecker (Spacecraft, PointGroup)

In addition to the constructor dependencies listed above sensors are associated with the Spacecraft via the AddSensor() operation provided by the Spacecraft class. CoverageChecker then accesses sensor(s) and their field of view via a Spacecraft object containing said sensor(s), providing a sensor ID to identify the correct sensor (current version supports only 1 sensor per spacecraft, and hence sensor ID can be ignored).

Finally, there is one other class of interest. Propagator and CoverageChecker each create a local copy of the Earth class; this class is primarily used to rotate vectors from an inertial frame (+X towards First Point of Aries) to an Earth-fixed frame (+X is 0 latitude, 0 longitude).

### Propagation & Coverage Without Interpolation

This shows the key processing loop in the case where coverage checks are done at the same rate as the spacecraft state is being propagated. The key steps are to

1. Propagate the spacecraft state up to the start time. This may involve a large ‘jump’, and is appropriate when drag effects are to be disregarded.
2. Loop until the end time; the date is a Julian date, which is expressed in ***days*** from a standard reference time.
   1. Check coverage.
   2. Advance the time and propagate orbit to that time. The step size is measured in ***seconds***.
   3. Compute latitude longitude and height.

prop->Propagate(\*date); // propagate to start time

while (date->GetJulianDate() < ((Real)startDate + 1.0)) // 1 day mission

{

// Compute points in view at time zero!

coveredPoints = covChecker-> CheckPointCoverage();

// Propagate

date->Advance(stepSize);

prop->Propagate(\*date);

// Compute lat., lon., and height of s/c w/r/t the ellipsoid

Real jDate = sat1->GetJulianDate();

Rvector6 cartState = sat1->GetCartesianState();

Rvector3 inertialPosVec(cartState(0), cartState(1),cartState(2));

Rvector3 latLonHeight = earth->InertialToBodyFixed(inertialPosVec,

jDate, "Ellipsoid");

}

The coveredPoints variable contains a list of point indexes for all the points of interest visible at the time that coverage is being checked.

### Propagation & Coverage With Interpolation

This option is used when the coverage checker’s event detection needs to take smaller steps than the orbit propagator. The Spacecraft sat1 provides a “time to interpolate” function that determines if interpolation is feasible. The inner loop will give the coverage checker the time to interpolate to when accumulating data, then advance the interpolation time.

prop->Propagate(\*date);

while (date->GetJulianDate() < ((Real) startDate + 1.0)) // 5.0))

{

date->Advance(stepSize);

prop->Propagate(\*date);

propTime = date->GetJulianDate();

// Interpolate when and if needed

if (sat1->TimeToInterpolate(propTime, midRange))

{

while (interpTime < (propTime - midRange))

{

loopPoints = covChecker->

AccumulateCoverageData(interpTime);

interpTime += interpolationStepSize/

GmatTimeConstants::SECS\_PER\_DAY;

}

}

}

In this scenario the interpolation step size is expected to be substantially smaller than the propagation step size. One second for interpolation and 1 minute for propagation is a plausible scenario.

# System Structure

The previous sections describe the high-level behavior of the propcov C++ library. This section documents the internal structure and highlights key functions and data structures contained within this subsystem. The next section diagrams the class dependencies, the following section documents the Propagator and Spacecraft, and the one after that documents the Coverage Checker. The detailed documentation includes the classes and their responsibilities, a list of key data structures, and a list of key functions.

## Class Dependencies

Spacecraft

CoverageChecker

Sensor

Attitude

ConicalSensor

DSPIPCustomSensor

OrbitState

Propagator

1,..n

Interpolator

AbsoluteDate

NadirPointingAttitude

1,..n

LaGrange

Interpolator

PointGroup

Earth

GMATCustomSensor

Other classes not in diagram:

* LinearAlgebra
* TATCException
* Dependencies of the DSPIPCustomSensor (8 classes)

In total there are 25 classes (including the 8 dependencies for the DSPIPCustomSensor) and [6 legacy classes](#_Legacy_code). The diagram above shows the key dependencies between components. The green shading shows the components that implement the main functions of modeling the spacecraft, propagating the spacecraft state, and identifying when points are within a sensor’s field of view. The blue shading indicates the models used by these major functions. The functionalities offered by the boxes with the orange outline are not in active use (and possibly unverified). Utilities such as vector and matrix arithmetic are not shown on this diagram.

Note that the class Sensor has three subclasses providing 3 different models of sensor field of views. The conical and rectangular fields of view are self-explanatory, a custom field of view allows the FOV perimeter to be defined by an arbitrary set of points. The following sections provide tables detailing each class’ responsibilities and explanatory text for key data structures.

## Propagation & Spacecraft

This section describes the responsibilities of each class used to model the spacecraft and its state, including the propagation of that state over time. It also lists the key functions and data structures used in this modeling.

### Class Responsibilities

| **Class** | **Responsibility** |
| --- | --- |
| Propagator | Propagates spacecraft state to a requested time using a J2 Analytical propagator. The default physical constants are set to that of spacecraft orbiting Earth. A Propagator object is initialized with a pointer to a spacecraft, from which the orbit-state is read & **modified**. Drag effects can be optionally considered by setting a flag, but this part is erroneous and needs revision. |
| Spacecraft | The Spacecraft class is a container for objects related to the spacecraft, including abstractions such as orbit and attitude, algorithms such as the LaGrange interpolator, or models of objects such as sensors.  The spacecraft class provides operations to access the state of its contained objects, and to do computations based on that state. Note that some of the containments are pointers to objects (e.g., orbitState, orbitEpoch), the objects which can be modified outside the Spacecraft object.  A key part of this spacecraft state that is maintained is the rotation matrix from the Nadir pointing reference frame to the body frame. This matrix is computed from user-set Euler angles & Euler sequence.  Another example is that the CoverageChecker calls Spacecraft’s CheckTargetVisibility operator, which rotates an input target-vector to the sensor frame and then calls the sensor to check whether the target is in the field of view.  The current implementation of the Spacecraft class has been verified with maximum of one sensor attachment. |
| Sensor | This class models a sensor. The Sensor class maintains knowledge of the sensor’s orientation relative to the spacecraft body and has a virtual-function (which must be defined in the child classes) to determines if a point is within the sensor field of view. It also defines a max-excursion angle which is the maximum cone angle corresponding to the sensor FOV (FOV could be of any shape).  There are three subclasses of Sensor. A conical sensor’s FOV is defined by a constant cone angle; ~~a rectangular sensor’s FOV is defined by angular width and angular height~~, both of which are symmetric around the boresight; and a custom sensor’s FOV is defined by an arbitrary set of points that are defined by cone and clock angle around the sensor frame’s +z axis.  For nadir pointing instruments the boresight axis is aligned with the spacecraft +z axis, and the body to sensor rotation is generally defined as the 3x3 identity matrix or an equivalent representation (e.g., quaternion or Euler angles). The rotation is to be specified by means of Euler angles and sequence. The rotation matrix rotates the coordinate system (See <https://mathworld.wolfram.com/RotationMatrix.html>). I.e., by performing R\_SB \* vecScBody, the representation of the vector in the sensor body frame is found. (R\_SB is the rotation matrix from the spacecraft-body frame to the sensor frame and vecScBody is the vector in the spacecraft-body frame.)  The Sensor class provides a CheckTargetVisibility() method which is implemented by each of the subclasses. This function determines if a vector (which must be rotated into the sensor frame to make this test valid) is inside the field of view or not. For cone ~~and rectangular~~ sensors these involve simple inequality tests, for the custom sensor a sophisticated line crossing algorithm is used.  The class also includes utilities to convert coordinates between different coordinate-representations (cone/clock, right-ascension/ declination, unit-vector, stereographic). |
| ConicalSensor | This is a subclass of the Sensor class. It can be used to evaluate the presence/absence of a point-location in a sensor FOV (conical shape). The target location must be expressed (clock/cone) in the Sensor frame.  The class relies upon the CheckTargetMaxExcursionAngle function inherited from the Sensor class to perform a simple check. |
| GMATCustomSensor  (originally called CustomSensor) | This is a subclass of the Sensor class. It can be used to evaluate the presence/absence of a target point-location in a sensor FOV (using the CheckTargetVisibility function). The target location must be expressed (clock/cone) in the Sensor frame.  The sensor FOV is to be described in the form of a spherical polygon, with a set of vertices (represented by clock/cone angles in the Sensor frame). A target point-location (represented by clock/cone angles in the sensor-frame) can be evaluated to be in/out of the FOV. The algorithm first checks if the target can possibly be in view by comparing the target cone-angle with the max-excursion angle. It performs stereographic projection of the polygon vertices and the target. A bounding box is formed, and target is evaluated to be within/out of the box. If within the box it then performs the even/odd rule check for ray-intersections to evaluate the presence/absence of the target in the sensor FOV.  Using the CheckRegionVisibility function, a region represented by vertices of a spherical-polygon can be evaluated to be completely in/out of the sensor FOV. |
| DSPIPCustomSensor | The DSPIPCustomSensor has an identical purpose as that of the GMATCustomSensor. It is subclass of Sensor and can be used evaluate the presence/absence of a target point-location in a sensor FOV (using the CheckTargetVisibility function). Either the DSPIPCustomSensor or the GMATCustomSensor can be used for evaluating point-coverage of a spherical-polygon.Differences with respect to GMATCustomSensor class:The DSPIPCustomSensor instantiation requires an additional input (along with cone/clock angles of vertices of the spherical polygon describing the Sensor FOV) of a point (cone/clock angles expressed in the Sensor frame) known to be contained within the Sensor FOV.This class defines two frames for the Sensor: Initial frame (**Sensor** Frame), and a Query Frame (alternate Sensor frame in which Queries are conducted.) The target point must be supplied in the Query fame and not the initial **Sensor** frame. The DSPIPCustomSensor is dependent on several other classes and together they are placed in a separate folder called ‘polygon’ inside the ‘propcov-cpp’ folder. More details of these classes can be found in the [Miscellaneous section](#_SphericalSensor). |
| Attitude | This is a pure-virtual base class used to model the spacecraft attitude state. It relies on the child-classes to implement the calculation of rotation matrices to rotate coordinate systems. |
| NadirPointingAttitude | NadirPointingAttitude is a subclass of Attitude that models the spacecraft in the Nadir-pointing coordinate frame. The main responsibility of this class is to compute the rotation from an (Earth) inertial/body-fixed frame to the nadir pointing reference frame (see [here](#_Propcov-Cpp_coordinate_systems)) using the input (spacecraft) state-vector (position and velocity). Note that there are two possible nadir-pointing attitude states which can be modeled using this class depending on whether the input spacecraft state is in inertial frame or body-fixed frame. |
| Interpolator | TBD |
| LaGrangeInterpolator | A subclass of Interpolator that computes interpolated values for arbitrary vector valued functions of a scalar independent variable. In this case the independent variable is time and the dependent vectors are position and velocity. |
| Earth | The Earth class models the Body-fixed (Body=Earth) and vector conversions relating to this frame. It provides for the following functions:   * Compute rotation matrix, or to rotate a vector from inertial to Earth-fixed frame. * Convert Earth-fixed vectors between Cartesian, Spherical and Ellipsoid representations. * Geocentric to geodetic coordinate conversions. * Calculation of sun-vector in body-fixed frame. * Compute rotation matrix, or to rotate a vector from body-fixed to topocentric. |
| OrbitState | OrbitState contains the spacecraft position and velocity (in Inertial coordinate frame), which can be set and retrieved as either Keplerian or Cartesian elements.  The default usage is with units of the Cartesian coordinates in km, km/s and Keplerian elements in km and radians. |
| AbsoluteDate | This class maintains a representation of date and time. The time can be set or retrieved as either a Gregorian date (year, month, day, hours, minutes and seconds) or a Julian date (days from a standard reference point), and it allows the date and time to be advanced by a number of seconds. This number may be negative to indicate movement backwards in time. |

### Key Data Structures

The data structures associated with the above classes tend to be scalar, vector or matrix member data, or references to other objects. The exceptions are:

* GMATCustomSensor, which contains several arrays related to the points that define the FOV boundary and for determining whether a point is in the field of view
* Interpolator, which contains arrays of values for independent (scalar) and dependent (vector) variables to be interpolated.

### Key functions

The key functions for propagation and spacecraft are:

Propagator

* Propagate() – this function calls PropagateOrbitalElements() and adds the option to model the effect of atmospheric drag by calling ComputePeriapsisAltitude()
* PropagateOrbitalElements() – this function propagates the Keplerian elements (a, e, i, RAAN, argP, MA), using the two-body problem with the addition of the J2 perturbation.
* ComputePeriapsisAltitude() – computes values needed in drag modeling

Spacecraft

* CheckTargetVisibility() – the implementation of this function is simple, it calls the CheckTargetVisibility() function in the Sensor class for a given sensor. The Sensor function in turn determines if a point is in its field of view.

## Coverage

Coverage evaluation determines if set of ground-points are within/out of sensor/spacecraft FOV.

### Class Responsibilities

|  |  |
| --- | --- |
| Class | Responsibility |
| CoverageChecker | This class checks for point coverage. The class is a reduced version of 'CoverageCheckerLegacy'. While the legacy version includes functionality to generate reports, this class only checks for point-coverage.  The CoverageChecker is instantiated with pointers to PointGroup object and a Spacecraft object. The point-group contains list of points which are to be checked for coverage calculations. The spacecraft may contain sensor, in which case coverage is evaluated for the sensor FOV or if no sensor the coverage is evaluated for the spacecraft (horizon-test is performed). There is room to expand to multiple sensors per spacecraft, but currently only 1 sensor per spacecraft is allowed.  The primary functions utilized are the overloaded functions CheckPointCoverage(.). First the CheckGridFeasibility(.) function is invoked to (1) determine if spacecraft and point are on the same hemisphere (2) if 1 is true, horizon check is performed. The above tests check the feasibility of point being covered. If feasible, the point is evaluated to be within/out of the sensor FOV. |
| PointGroup | PointGroup maintains a user defined or an automatically generated set of points (both Cartesian and Spherical) on the surface of the central body. These points are accessed by an integer point ID and represented in terms of longitude and latitude or of a position vector expressed in the central body’s rotating coordinate frame (body-fixed coordinates).  Points may be set on input (or) computed in the class based on:   * Specified number of points within a region (specified by Lat/Lon bounds). * Specified angle resolution within a region (specified by Lat/Lon bounds).   Note: Latitudes must in the range of -90 deg to +90 deg and longitudes must be in the range of -180 deg to +180 deg while inputting points. Class **cannot** handle longitudes in range of 0 deg to 360 deg. |

### Key data Structures

The key data structures for coverage checking all reside in the CoverageChecker class. They are supported by the class members in the PointGroup, VisiblePOIReport, and IntervalEventReport; all of which are containers with little or no processing beyond setting and getting data. These data structures are:

* sc – Spacecraft from which the state, date, sensors are obtained.
* pointGroup – is a pointer to the pointGroup being analyzed. The constructor sets this pointer from the input parameter ptGroup.
* pointArray – is an array of unit vectors representing the position of each point in pointGroup in the central-body-fixed reference frame.
* feasibilityTest – is an array of bools which is set by the GridFeasibility(.) function.

### Key functions

The key function for coverage checking is the CheckPointCoverage(.). This in turn invokes the spacecraft CheckTargetVisibility(.) function, which in turn invokes the Sensor CheckTargetVisibility(.) function.

# Attitude Mathematics

This section is intended to clarify how attitude and other rotations (e.g., from spacecraft body to sensor coordinate frame) are used in Propcov-Cpp. This incorporates two sections of a larger set of developer notes on attitude. The first provides a general high-level background on attitude. Its intent is not to be comprehensive, but to guide developers in understanding the basics of the problem being solved. As an example, it describes key coordinate systems, but it does not cover the details of converting between representations such as quaternion and direction cosine matrix.

The second part of this section discusses the specifics of Propcov-Cpp; what coordinate frames are used and where the attitude math lives in the code.

## Attitude Background

Any background material should start with definitions, and this document does not vary from that rule. We will then dive into two common representations of attitude, the direction cosine matrix (DCM) and the quaternion. The background material will end by describing some of the most used coordinate reference frames for GMAT, and more importantly *why* they are commonly used.

### Definition

First the definitions. Attitude obviously has other meanings unrelated to aerospace, but we will just look at some dictionary definitions from online.

From Merriam-Webster.com:

*5****:*** *the position of a craft (such as an aircraft or spacecraft) determined by the relationship between its axes and a reference datum (such as the horizon or a particular star)*

From vocabulary.com:

*3: position of aircraft or spacecraft relative to a frame of reference (the horizon or direction of motion)*

From dictionary.com

*3: Aeronautics . the inclination of the three principal axes of an aircraft relative to the wind, to the ground, etc.*

All of these definitions give you the gist of what attitude is about, none of them exactly matches the mathematical definition we will be using. The mathematical formalism is to model attitude as the rotation from a reference three-axis coordinate frame to a three-axis frame fixed to an aircraft or spacecraft. The selection of reference frame will depend on the type of mission. Space science missions are more likely to reference an inertially fixed frame, while earth science points the instruments downward, and pick a reference frame that rotates (for those who are not beginners, it is pitching at 1 revolution per orbit) to keep the instrument pointing downward.

Another aspect of attitude modeling is that when you are modeling attitude dynamics (the response of the aircraft or spacecraft to the torques being exerted on it), the equations of motion are usually written with respect to an inertial frame. So, an Earth-pointing satellite may estimate its attitude with respect to an inertial frame using the equations of motion, do the mathematics to compute the rotation to a non-inertial downward-pointing reference frame, and control the spacecraft to the desired orientation with respect to the downward pointing frame.

So far, we’ve defined what attitude is. There are several ways to represent attitude, which are discussed in section 12.1 of [Wertz 1978]. Section 4.3.2 of the GMAT Mathematical Specifications [GMAT 2018] specifies conversion between these representations. There are several parameterizations of attitude presented in Wertz and the math specs; for now, we will concentrate on the ones that are the *direction cosine matrix* and the *quaternion.*

Direction cosine matrices will be familiar to those who have taken introductory linear algebra, quaternions tend to be taught first in advanced mechanics classes or in computer graphics courses. The next two sections will discuss how rotations are used in actual flight dynamics applications using cosine matrices and quaternions, respectively.

### Properties of Rotation (Attitude) Matrices

We will now look at the attitude as the rotation from inertial to body frame. If it’s represented by a cosine matrix, that can be written as **A** or . Quaternions are generally written as **q**, as they are primarily used in propagating the kinematics or dynamics of the attitude over time. The use of **boldface** indicates a vector or matrix. For this section we will discuss rotation matrices (another name for cosine matrices), similar properties exist for quaternions, which are discussed in the next section.

So, enough background, we can now look at what we can do with attitude matrices. The first application is to *express vectors in different reference frames.* For example, if one wants to know the location of the sun relative to the spacecraft, one would use the spacecraft to sun vector expressed in the spacecraft body frame. However, most models of solar ephemerides (position and velocity) are modeled in an inertial frame that is fixed in space.[[1]](#footnote-1)

So how do we express the inertial sun vector we have to the body sun vector we want? You simply multiply it by the cosine matrix representing the rotation from inertial to body. Let represent the sun vector in the inertial frame. Then

= **A**

Where **A** is the attitude matrix.

Note that there are two types of rotation matrices which can be constructed (and are transpose of each-other): (See <https://mathworld.wolfram.com/RotationMatrix.html> for details.)

1. Rotation matrix to rotate the coordinate system and to use it to find vector representation in other coordinate systems.
2. Rotation matrix to rotate vector relative to fixed axes.

The second useful property of rotation matrices is that you can compose two (or more) rotations via matrix multiplication. For example, sensors have their own coordinate frame in which the field of view is defined. This frame may or may not be aligned with the body frame, generally it will not be. In this case, let the rotation from body to sensor frame be labeled . Then the rotation from inertial to sensor frame is

**=** .

Note a good check of if you are composing things correctly is that the first rotation (the matrix to the right) rotates into the body frame, and the second rotation is from the body frame to sensor frame. If all the rotations in an equation follow this protocol, then the outermost subscript letters will tell you about the frames for the composition of two or more rotations.

It is also worth noting that in general will vary over time, while the body to sensor frame rotation generally won’t.[[2]](#footnote-2)

Rotating a vector then becomes

**= =**  = =

This gives a sequence of equivalent representations using substitution and the associative property to arrive at the composite rotation.

The third useful property of rotation matrices is that they are easily invertible; cosine matrices are inverted by transposing them, or  **=** . If **A** represents the rotation from a reference to body frame, then its inverse represents the opposite rotation from body to inertial, or

**=**

To sum up the three properties

* Vectors are expressed in a new coordinate frame by multiplying the vector by the rotation matrix from the initial frame to the new one.
* Multiplying two or more rotation matrices represent successive rotations between coordinate frames.
* The rotation back to the original frame is possible by multiplying the vector with the transpose of the original rotation matrix.

### Using Quaternions: The Basics

This section is a placeholder for future writing. We don’t use quaternions in Propcov-Cpp.

### Coordinate Frames for Spaceflight

Now that we have seen the basic properties of rotation matrices, let us look at the frames that are of interest, starting with the inertial frame.

#### Inertial Reference Systems

The **Earth-Centered Inertial (ECI)** frame is defined by the following axes

* The origin is at the center of the Earth[[3]](#footnote-3)
* +Z points to the spin axis of the earth
* +X points to the vernal equinox (aka the First Point of Ares); the point where the Earth equatorial plane intersects the Earth’s orbital plane, and the sun is moving from southern to northern hemisphere on the first day of spring.
* +Y completes right hand system (**Y = Z x X**)

The same frame is also labeled *Geocentric Inertial (GCI)* in some applications.

Now in the best of all possible worlds, this coordinate system would be truly inertial in that it does not rotate over time. Unfortunately, we do not live in the best of all possible worlds, and the ECI (slowly) rotates.[[4]](#footnote-4) The solution we use is to take a snapshot at a given time (also called the *epoch*) and arbitrarily fix the ECI axes based on that epoch. GMAT uses the **J2000** frame as the standard inertial coordinate system for calculations; more formally GMAT calls it Mean J2000 Equatorial (MJ2000Eq) system, or Mean of Date Equatorial at the J2000 Epoch, which in turn is around noon on January 1, 2000; the exact time depending on the time system being used.[[5]](#footnote-5)

Another inertial is the **International Celestial Reference Frame** (ICRF), a newer frame that is defined by the measured positions of extragalactic sources (mainly quasars)[[6]](#footnote-6), which creates a more accurate system. This system is *not* an ECI system, its origin is at the Sun-Earth barycenter[[7]](#footnote-7).

There are many other inertial reference systems to be aware of when using GMAT, but we will stop here before we get too far away from the attitude world.

#### Central Body Fixed Coordinate Systems

The next coordinate system of interest is one that is fixed to the body the spacecraft is orbiting. This system is used when

* The spacecraft is looking for/at specific objects on the ground, for example ground stations to which telemetry is sent.
* Modeling is dependent on the specific orientation of the earth; for example, an accurate gravity model depends on the shape of the earth, which is not a smooth sphere. Which way Mount Everest is pointing in inertial space will matter in computing the Earth’s gravitational effect on the spacecraft.

Such coordinates can be defined for any central body, here we will just discuss the **Earth-Centered, Earth-Fixed (ECEF)** reference frame. It is similar to ECI, but instead of staying fixed in inertial space it rotates in lockstep with the Earth. The frame is defined as follows:

* The origin is at the center of the Earth
* +Z points to the spin axis of the earth
* +X points to the intersection of the Equator and the Prime Meridian
* +Y completes right hand system (**Y = Z x X**)

Because ECEF and ECI share the Z axis, the rotation from ECI to ECEF reduces to a rotation around the +Z axis. The angle associated with this rotation is the *Greenwich hour angle*. The rotation matrix associated with the Greenwich Hour Angle G is

Where “F”in the subscript represents the Earth-fixed frame and “I” the inertial frame.[[8]](#footnote-8)

#### Reference Coordinate Systems

In principle, many frames can be used as reference frames for spacecraft, they can fall into two categories. The first is other inertial frames, for example frames centered on other bodies than the Earth or fixed at different reference times. The second category is orbit-referenced frames, which are constructed from the spacecraft’s position and/or velocity and have the origin at the center of the spacecraft.[[9]](#footnote-9) These frames typically have one of the axes pointing down towards the central body, although there are a variety of definitions possible.[[10]](#footnote-10) From here on, we will use Earth as the central body and assume the reader can generalize this to other bodies.

The next step, of course, is to define “down”. One would think there would be a unique definition, and if the Earth were a perfect sphere a vector from the spacecraft to the center of the Earth would uniquely define “down”. But since it isn’t, we define two versions of down, the **nadir vector** that points to the center of the Earth, and the **local vertical**, which is perpendicular to the surface, which is defined by a plane tangent to the ellipsoid that defines the Earth. A **nadir-pointing** coordinate system has an axis pointing to the center of the Earth, and a **geodetic** system has an axis aligned with the local vertical. The former is simpler, so we will focus on that.[[11]](#footnote-11)

A typical nadir pointing system is the LVLH Earth-Pointing system defined in AI Solutions Free-Flyer:

* The +Z axis points to the nadir (equivalently, is aligned with negative position vector)
* The +Y axis is the negative orbit normal defined by position and velocity ( = **-RxV** ), where R is the spacecraft position vector.
* The +X axis completes the right-handed system

Search the STK or AI Solutions web sites for other trajectory-based reference frames.

These systems are used as reference for attitude systems; for example, a satellite might control attitude to keep the body frame aligned with a geodetic frame, so that the sensors are looking along the local vertical to observe the Earth. Control laws would then be written with respect to the rotation from reference to body frame, although the actual kinematics or dynamics would be propagated in an inertial frame.

## Propcov-Cpp Use Cases

Propcov-Cpp looks at points on the Earth’s surface from two perspectives. The first is whether the spacecraft can possibly be seen from the point of interest. This is not a full test of visibility, it’s a simple check that will eliminate points on the opposite side of the Earth.

The second perspective is whether a point (grid-point/ Ground-station) is actually in a sensor field of view. Unlike the first test, this requires attitude information and the orientation of the sensor to the spacecraft body frame.

The next sections define the coordinate frames used in Propcov-Cpp, then lays out a use case for each of the two perspectives in mathematical terms.

### Propcov-Cpp coordinate systems

Propcov-Cpp defines the following frames:

* **Inertial (I)** – uses the J2000 Mean Equatorial frame as inertial reference
* **Fixed (F)\*** – uses the Earth-Centered, Earth-Fixed frame to account for Earth’s rotation. In the code this is also referred to as the *Central Body Fixed* frame.
* **Nadir-Pointing (N)** - This is defined as follows:
  + The +Z axis points to the nadir (equivalently, is aligned with negative position vector).
  + The +X axis is the negative orbit normal defined by position and velocity ( **n ̂ = -RxV** ), where R is the spacecraft position vector.
  + The +Y axis completes the right-handed system.

The Nadir-pointing frame can be constructed to be aligned to *either* the spacecraft velocity vector in **Inertial** or the **Fixed** frame.

* **Body (B)\*** – frame aligned with the spacecraft body; this frame is usually aligned with the nadir-pointing frame, but an off-nadir alignment can also be specified.
* **Sensor (S)** – frame attached to sensor where +Z represents the boresight or origin for modeling the field of view.

**\*** Several functions/variables with ‘body’ in it may refer to either the Earth or the Spacecraft.

A rotation is represented by the 3x3 matrix **R\_XY**, which represents a rotation from frame Y to frame X. This convention is used in variable naming in the code.

A counterclockwise rotation is regarded as positive rotation, while clockwise rotation is negative rotation.

### Referencing targets on Earth (attitude independent use case)

There are two types of attitude independent visibility checks that are done during coverage evaluation. The first is a dot product check to eliminate points that are blocked by the bulk of the Earth. If the dot product of the spacecraft position and the ground point’s position is less than zero (i.e., the spacecraft and the ground-point are on opposite hemispheres, where the hemisphere is formed by the plane defined by the unit-normal along the ground-point position-vector), the point can be eliminated.

The second visibility check computes the vector from the spacecraft to the ground point and uses it to determine whether the spacecraft is over the horizon when viewed from that point.

This is done in the routine CheckGridFeasibility(.), which loops through all points of interest and eliminates the obviously unfeasible before any real processing starts. This function takes the position in body fixed coordinates as input and iterates through the points of interest.

The common element of both checks is that the point of interest positions are in ECEF coordinates, while the spacecraft position vector is propagated in the inertial reference frame.

The solution is to rotate the inertial position vector to the Earth-fixed frame, using

In the interests of computational efficiency this is done outside the loop that iterates over all the points of interest. See CoverageChecker::CheckPointCoverage(.) for the details of the code.

### Rotating vectors to sensor coordinate frames (attitude dependent use case)

To express the satellite-to-target vector (in Fixed frame) in the Sensor frame, we do the following:

is the rotation matrix from Fixed frame to Nadir pointing frame.

is the rotation matrix from Nadir pointing frame to (spacecraft) Body frame. In nominal mission operations this is how the spacecraft attitude is defined.

is the rotation matrix from (spacecraft) Body frame to the Sensor frame.

This is done in the function Spacecraft::CheckTargetVisibility(.) overloaded function.

## References

{Wertz 1978] Wertz, James, editor. Spacecraft Attitude Determination and Control. D. Reidel Publishing Company, Dordrecht, Holland 1978.

[GMAT 2018] General Mission Analysis Tool (GMAT) Mathematical Specifications DRAFT, February 9, 2018.

<https://mathworld.wolfram.com/RotationMatrix.html>

# Miscellaneous

## Clock/ Cone angles representation of a point-location

Clock, cone angles are used to express the point-location on a unit-sphere, and several calculations involving checking if the point-location is in/out of sensor FOV is done using this representation. The relationship of the clock/cone angle representation with the “standard” right-ascension (RA)/ declination (dec) representation is in the ConeClocktoRADEC function in the Sensor class.

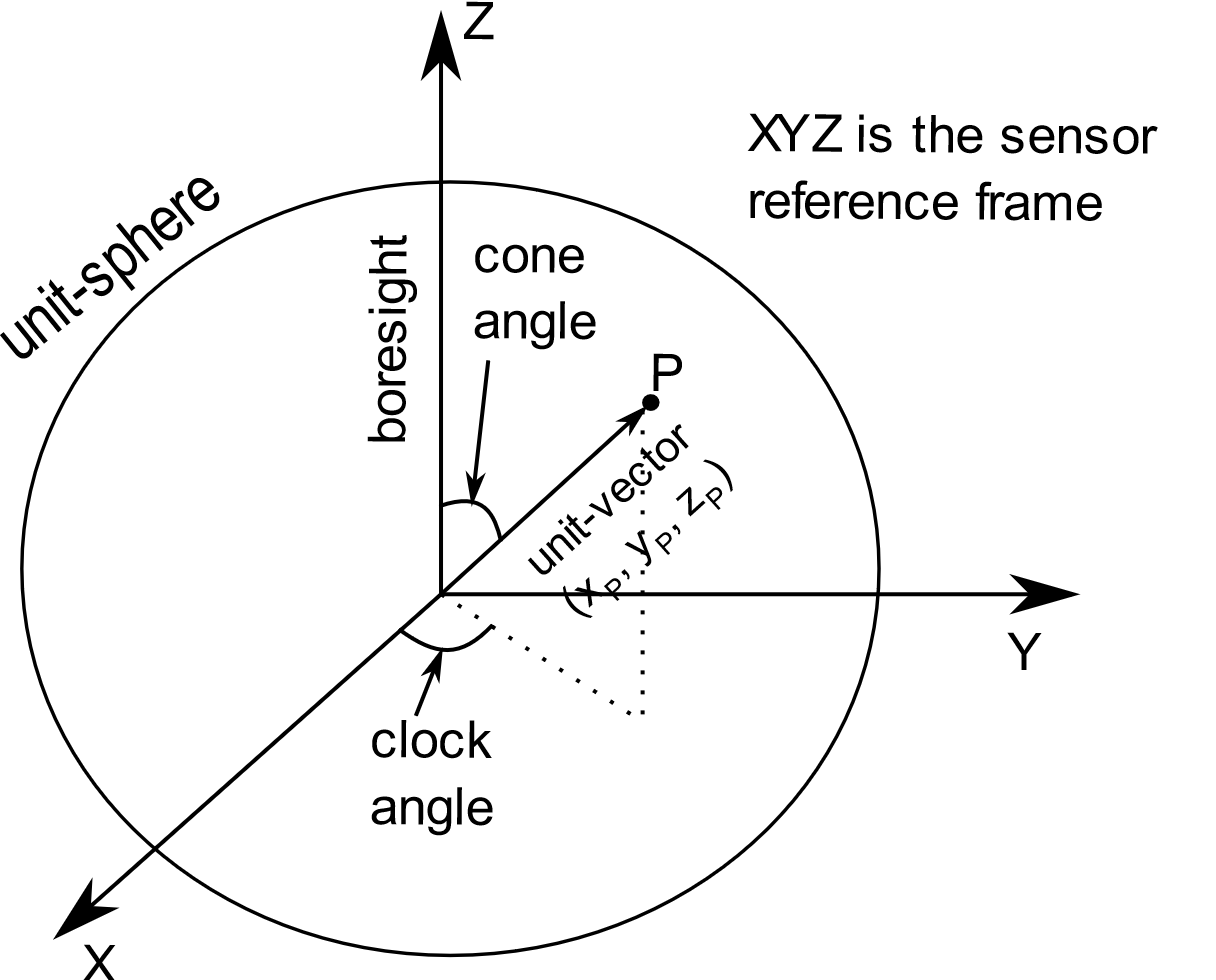


Illustration of cone/ clock angles

The sensor boresight axis is assumed to be pointing along the z-axis of the Sensor frame.

## DSPIPCustomSensor description

The DSPIPCustomSensor class is based on a point in spherical polygon algorithm described in the article: R. Ketzner, V. Ravindra and M. Bramble, 'A Robust, Fast, and Accurate Algorithm for Point in Spherical Polygon Classification with Applications in Geoscience and Remote Sensing', Computers and Geosciences, under review.

In the above article, the algorithm is described and compared to the ‘GMAT CustomSensor’ algorithm which is the same as the point-in-polygon algorithm implemented in the GMATCustomSensor class. Compared to the GMATCustomSensor class, the DSPIPCustomSensor has been shown to yield orders of improvement in runtime (for FOVs defined by spherical polygons of several vertices) and shown to be more accurate.

In the OrbitPy library, the coverage method can be specified to be ‘DirectSphericalPointInPolygon’ which corresponds to usage of the DSPIPCustomSensor class, or ‘ProjectedSphericalPointInPolygon’ which corresponds to usage of the GMATCustomSensor class.

The dependencies of the `DSPIPCustomSensor class are listed below.

|  |  |
| --- | --- |
| Class | Description |
| Polygon | The Polygon class is a pure virtual class which defines the interface for spherical polygons. This interface requires all child classes to implement a function which takes query point coordinates as input and returns an integer value representing point inclusion status in the polygon. The Polygon file also includes a util namespace defining some useful function used across the program (spherical/cartesian coordinate conversion, file i.o., etc.). |
| Preprocessor | The preprocessor class is a pure virtual class which defines the interface for preprocessing algorithms. This interface requires all child classes to implement a method which returns a list of indices representing a subset of edges to be examined as a function of the query point coordinates. |
| SlicedPolygon | A child class of Polygon which implements the proposed point in spherical polygon algorithm. The algorithm can be run with or without preprocessing. To run with preprocessing, a Preprocessor object must be added using the addPreprocessor method. |
| Edge | A utility class to simplify representation of polygon edges across the program. The edge class is constructed with two coordinate points, and implements useful functions such as checking whether a given query point meets necessary strike or the hemisphere check with respect to the edge. |
| SliceArray | The proposed preprocessing algorithm described in the paper. The class is constructed using an array of vertex longitudes in the query frame. The preprocess() method is used to run the preprocessing routine after construction. The getEdges takes method takes the coordinates of a query point as input and returns the subset of edges classified in the leaf slice containing the query point. |
| SliceTree | This class implements the old version of the preprocessing algorithm using the recursive special subdivision technique. |
| PointInPolygon | Utility class for running the algorithm from the command line and generating point inclusion and runtime report output files. |
| GMATPolygon | Used (in tests) for comparing the GMAT CustomSensor algorithm with the Direct Spherical Point In Polygon algorithm. To be removed. |

## Legacy code

The following classes from the original TAT-C project are not in active use:

1. CoverageChecker class in the CoverageCheckerLegacy hpp & cpp files
2. VisiblePOIReport
3. IntervalEventReport
4. VisibilityReport
5. KeyValueStatistics
6. RectangularSensor

Documentation on CoverageChecker (Legacy) and related classes can be found in the document ‘Orbit & Coverage design doc 1890325 v1.2.doc’.

CoverageChecker (Legacy)

1,..n

PointGroup

Visible POI

Report

IntervalEventReport

1. This is not strictly true, as we will see in the next section. But it’s close enough. [↑](#footnote-ref-1)
2. If a sensor is mounted on a component that rotates with respect to the main body of a spacecraft (such as a solar array), then the body to sensor frame will vary. [↑](#footnote-ref-2)
3. Coordinate systems parallel to ECI can be centered on other celestial bodies, or even points in space such as barycenters or libration points. For earth orbiters, ECI is sufficient, and that is where this version of the documentation will stay. [↑](#footnote-ref-3)
4. The most commonly modeled effect is precession of the Earth’s spin axis in inertial space. Other effects to consider are nutation (wobble) and changes to the Earth itself due to slosh of the liquid core or melting Greenland and Antarctic ice caps. [↑](#footnote-ref-4)
5. Time systems are a whole other issue that won’t be discussed here. Their largest importance is in accurately modeling the positions and velocities of celestial bodies in the solar system. See Section 1 of the GMAT math specs. [↑](#footnote-ref-5)
6. https://en.wikipedia.org/wiki/International\_Celestial\_Reference\_Frame [↑](#footnote-ref-6)
7. A barycenter in this context is the center of mass of two or more bodies that orbit one another. [↑](#footnote-ref-7)
8. There is a whole lot involved in converting time from Julian dates to a Gregorian date and time, all of which we will ignore here. [↑](#footnote-ref-8)
9. Technically, they are trajectory-referenced, orbit-referenced is the subset for a spacecraft [↑](#footnote-ref-9)
10. See <https://ai-solutions.com/_help_Files/attitude_reference_frames.htm> and <http://help.agi.com/stk/index.htm#stk/referenceframesvehicle.htm%3FTocPath%3DGetting%2520Started%7CTechnical%2520Notes%7C_____3> to see some examples. Note that AI Solutions defines LVLH with +Z aligned with the negative position vector, and STK defines +X to be aligned with the position vector. [↑](#footnote-ref-10)
11. Finding the local vertical involves finding the tangent plane, which is done through computing the derivative of the ellipsoid. This is left as an exercise to those who 1) have taken multivariate calculus, and 2) haven’t had years to forget what they learned. [↑](#footnote-ref-11)