Documentation for Astrometry Classes

G. M. Bernstein

Dept. of Physics & Astronomy, University of Pennsylvania

`garyb@physics.upenn.edu`

1. Dependences

All routines described here are placed in the `astrometry` namespace. All exceptions are thrown as the `AstrometryError` class, which is derived from `std::runtime_error`.

The coordinate routines involve frequent manipulation of 2- and 3-dimensional vectors and matrices. These are implemented by use of the "`Small`" feature of Mike Jarvis's *TMV* linear algebra package, which uses templates to hard-wire the dimensionality and avoid a lot of overhead. *Astrometry.h* starts with `typedef`'s to give local type names `Vector2, Vector3, Matrix22, Matrix23,` etc., to vectors and matrices of the given types. Moving to another vector/matrix package should just involve changing these typedefs, as long as the package has properly implemented multiplication operators and element access with the same syntax as *TMV*. Also the *TMV* routine `setToIdentity()` is used a lot.

[**Note:** There is one gotcha in the `TMV::SmallMatrix` class, which is that there is no checking for aliasing, so expressions like `v = m * v` can fail.

The utility file `AstronomicalConstants.h` is included by `Astrometry.h` to define many constants used in astrometry and ephemeris programs. Of particular interest:

- `PI` and `TPI`$= 2\pi$ are defined and angular units `DEGREE, ARCMIN, ARCSEC` are defined. They give the size of each unit in radians. The constant `RadToArcsec` is the number of arcseconds in a radian.
- Time units `YEAR, DAY, MINUTE, SECOND` are defined in the units of orbital mechanics, giving each in units of a year. A `DAY` is defined as exactly 86,400 `SECOND`s and exactly 1/365.25 `YEAR`s. `MJD0` is the offset between Julian dates and modified Julian dates (MJD).
- The orientations of the ecliptic and invariable planes with respect to J2000 coordinates are defined.
- Other constants used by ephemeris programs include `GM, SolarSystemMass, EarthMass`.
- And a bunch of other astronomical constants, such as `AU` and `Parsec`, usually given in mks units.

2. Time

The class `UT` represents a time. There are many different kinds of time, and the time in the class is assumed to represent UTC. It can be initialized or set from different forms: JD, MJD, year/month/date/hr/min/sec, etc.

The `-` operator is overloaded to yield the interval between two times. The `+=` and `-=` operators are overloaded to allow time to be advanced/set back by a specified intervals. Time intervals are in units of `YEAR`s. Subtracting/adding times is done in such a way as to account for leap seconds, though I do not gaurantee success if one of the times is within a minute or so of the leap second itself.

2.1. Time systems

The time used in the JPL ephemeris is TDB (barycentric dynamical), which is within 0.002s of TT (Terrestrial time), which is a coordinate time at Earth's geoid. TT is a constant ($\sim$ 34 s) off from TAI=atomic time. UTC has leap seconds to keep it close to the Earth rotation time, so TT-UTC is a function of time. The list of leap seconds is maintained in the file *LeapSeconds.h* as the function `UT::TAIminusUTC()`.

2.2. Internal representation and accuracy

Times are represented internally by a double-valued UTC JD. Doubles have 52 bits mantissa, so JD should be accurate to 0.1 ms or better, which is <(TDB-TT).

2.3. I/O

The `read()` method will recognize any of the following formats. Any field can be represented by any number of digits. Fields are separated by any whitespace.

- *jd.dddd* —Julian date. Any number >10,000 or having a decimal point in its first argument is interpreted as a JD. No characters are read from the stream beyond the first field.
- *yr mo dd.dddd* —if the third field has a decimal point, it is interpreted as a fractional date and no further fields are read. Three fields must be present else the input stream is marked with `failbit`.
- *yr mo dd hh:mm:ss.ssss* —if the third field is an integer, a fourth field is expected in hours/minutes/seconds format, which is read using the conventions of the `hmsdeg()` subroutine.

Writing can be requested into either of the last two string formats, using the current `precision()` of the output stream for the number of chars past the decimal point in the last argument. The overloading stream operator `<<` uses the YMD format. Note that 1 second is $\approx 10^{-5}$ days.

3. String manipulations

The functions `dmsdeg` and `hmsdeg` return double-value numbers of degrees from input strings with the formats *dd:mm:ss.sss* or *hh:mm:ss.sss* respectively. Colons or spaces can separate the 3 fields. Leading whitespace is skipped. Any fields can be an integer or have fractional parts after a decimal point. Number of digits per field is not constrained. Seconds can be omitted, or minutes and seconds can be omitted. A leading minus sign will negate the entire value, no signs are allowed beyond the first field. There is no range checking on any of the values.

The functions `degdms` and `deghms` return formatted string representation of input degree-valued double-precision numbers. The output format of `degdms` is $[+-]$*ddd:mm:ss.sss* or $[+-]$*dd:mm:ss.sss* depending on whether there are ≥ 100 degrees on input. The output format of `deghms` is $[-]$*hh:mm:ss.sss*. The function call includes a specification of the number of decimal places in the seconds argument.

4. Spherical coordinate systems

`SphericalCoords` is an abstract base class for representing a location on the celestial sphere. Every instance of the base class represents a well-defined position on the sky independent of coordinate systems.

There are many derived classes, each tied to a specific 2-dimensional coordinate system on the sky, *e.g.* `SphericalICRS` for RA/dec coordinates in the ICRS system, `SphericalEcliptic` for ecliptic lon/lat coordinates. Every derived class instance can be initialized or set using a pair of (longitude, latitude) values (or a `Vector2` containing the 2 coordinates). **Coordinates are always assumed to be in radians, and the x or longitude coordinate comes first. However the conversions to/from strings and the overloaded stream operators will use units that depend on the coordinate system.** For example the lon (lat) will be printed out in sexagesimal hours(degrees) by the base-class implementation, to follow tradition in representing RA and declination.

The `distance` method gives the great-circle distance between any two `SphericalCoords` instances. This is well defined regardless of the coordinate systems of the two operands, since every `SphericalCoords` represents a definitive position on the celestial sphere.

Every `SphericalCoords` must implement the `duplicate()` method, which returns a pointer to a new object with the same meaning as itself. The duplicate is a deep copy.[1]

4.1. Coordinate transformations

Coordinate transformations never have to be done explicitly! One merely has to construct an instance of a derived class in the new coordinate system from a `SphericalCoords` in the old system. For example, to convert ecliptic coordinates into :

```
double eclipticLon=0.33;
double eclipticLat=-0.55;
double icrsLon;
double icrsLat;

SphericalEcliptic ecl(eclipticLon,eclipticLat);
SphericalICRS icrs(ecl);
icrs.getLonLat(icrsLon, icrsLat);
```

The constructors know how to do all the conversions! Or you can reset an existing instance using the `convertFrom` methods:

```
double eclipticLon=0.33;
double eclipticLat=-0.55;
SphericalEcliptic ecl(eclipticLon,eclipticLat);

SphericalICRS icrs;   /* initialized with some other value*/

icrs.convertFrom(ecl);
icrs.getLonLat(icrsLon, icrsLat);

Matrix22 partials;
icrs.convertFrom(ecl, partials);
```

In the last two lines, we use another generic feature of `convertFrom`: it can give you the partial derivative matrix of the conversion that you just did. This is used for many things, such as propagating positional uncertainties into a new coordinate system.

[1]An exception is when classes such as `Gnomonic` are created with `shareOrient=true`. In this case the duplicate and the original, share a pointer to an `Orientation` that neither owns.

4.2. Unit-vector representations

`SphericalCoords` do not store the longitude/latitude representation, they store the position in the selected coordinate system as a unit-length 3-dimensional `Vector3` object. Coordinate conversions are much faster and singularity-free when coordinates are stored this way. The `getUnitVector()` and `setUnitVector()` methods allow you access this representation of the coordinates. The meaning of the three-vector depends on the type of the derived class, just as the lon/lat values do, *i.e.* are they ecliptic, Galactic, ICRS, etc.?

You can also obtain the partial derivatives of the 3-dimensional representation with respect to the 2d ones, or vice-versa, with the `derivsTo2d()` and `derivsFrom2d` methods, respectively. The partial derivatives of a transformation between coordinate systems that are obtained from the `convertFrom()` method can be between the 2d or 3d representations of both systems. The dimension of the matrix that you give the method will signal the kind of derivatives that you want. For example if you hand the routine a reference to a `Matrix23`, you will get the partial derivatives of the new lon/lat coordinates with respect to the 3d direction cosines in the old coordinate system.

Note that a request for partial derivatives to/from a lon/lat system can generate an `AstrometryErrror` exception if the coordinates are at the pole, since there is a singularity. Other coordinate systems might throw exceptions if venturing to singular points or undefined regions.

4.3. Implementing a new coordinate system

The automated coordinate conversions are possible because the author of a new derived class must implement four protected methods. `convertToICRS` must give the unit vectors in the ICRS system (and also be capable of giving the partial derivatives of the ICRS 3-vector with respect to the native 3-vector). The `convertFromICRS` method must be able to set the native 3-vector given an ICRS `Vector3`. Hence the ICRS system serves as a *lingua franca* for all implemented coordinate systems.

Two other protected methods both called `projectIt` specify the maps in both directions between the 3d direction cosines and the 2d lon/lat coordinates for the implemented coordinate system. These methods are implemented in the base class using the common definition that the 2d coords are indeed the latitude and longitude in a system with pole along the $\hat{z}$ direction and coordinate origin on the $\hat{x}$ unit vector. However this can be overridden to allow the 2d coordinates to have any desired meaning.

Hence to define a new coordinate system, one derives a class from `SphericalCoords` and implements, at the least, the two ICRS conversion methods. The `projectIt` methods can optionally be implemented to have 2d coordinates that are not simply the lon/lat values with respect to the 3d system.

A new coordinate system must also implement the `duplicate()` method, usually simply by `return new XCoords(*this)` for some `XCoords` system.

4.4. Implemented coordinate systems

4.4.1. `SphericalICRS`

Coordinates are in the ICRS system. The `convertToICRS` and `convertFromICRS` are simply identity functions and the partial-derivative matrices are identity matrices. Latitudes and longitudes have their standard meanings, with longitude increasing westward. The implementation contains the `getRADec` and `setRADec` methods that simply alias the `LonLat` functions, so the user does not have to remember that RA is the longitude and declination is the latitude in this system.

4.4.2. `SphericalEcliptic, SphericalInvariable`

These are longitude/latitude systems with fixed rotations relative to the ICRS system, defined in the usual way with respect to the Earth's orbital plane and the invariable plane of the Solar System. The inclinations and ascending nodes of these planes relative to ICRS are taken from the *AstronomicalConstants.h* file.

Because rotation to a new pole is a very common coordinate transformation, there are `Vector3`-valued functions `rotateToPole()` and `rotateFromPole()` defined in the implementation file *Astrometry.cpp.* These will provide the new unit vector (and the partial derivatives matrix) for a coordinate system with its pole having the specified inclination and ascending node with respect to the ICRS system.

4.4.3. `SphericalCustom`

This is a latitude/longitude system with an orientation on the celestial sphere that is specified at the time of construction of the object. The constructor for `SphericalCustom` requires a reference to an instance of the `Orientation` class. This class is described in more detail below: it specifies the location of the system's coordinate origin on the sky, and the rotation angle of its axes.

On construction, if the default `shareOrient=false` is taken, the `SphericalCustom` instance saves its own copy of the `Orientation`. If you select `shareOrient=true`, only a pointer is saved, so the `Orientation` used during construction should not be altered or destroyed during the lifetime of all `SphericalCustom` coordinates that refer to it. This can save space and computation if you are manipulating many `SphericalCustom` instances with the same `Orientation`.

4.4.4. `Gnomonic`

The coordinate system is a tangent-plane (gnomonic) projection of the sphere about a reference point specified at the construction of each instance. As with `SphericalCustom`, the coordinate origin and the orientation of the tangent plane axes are specified by reference to an `Orientation` object. The `shareOrient` flag on construction has the same effect as for `SphericalCustom`. The difference is that the 2d coordinates are not actually longitude and latitude, but rather the Cartesian coordinates (ξ, η) of the projection of the point onto the tangent plane from the center of the sphere. In terms of the direction cosines $\{x, y, z\}$ of the point on the celestial sphere, the 2d coordinates are $\xi = x/z$, $\eta = y/z$. Note that the tangent plane 2d coordinates make sense only within $90°$ of the tangent point.

The `read` and `write` methods of `Gnomonic` override the base class such that both coordinates are input or output in degrees.

4.5. `Orientation` class

`Orientation` defines a new point of view of the celestial sphere, *i.e.* a rotation of the sphere. On construction one must specify a `SphericalCoords` that will be the z axis of the new viewpoint (the pole), and a double-valued position angle that will give the rotation angle of the (x, y) coordinate system with respect to the local ICRS meridian at the pole. "Position angle" or "PA" arguments are always in radians and give the angle of the y axis measured from north through east. Since east corresponds to increasing RA, this convention means that PA is the negative of the usual mathematical convention of a rotation as being positive as the x axis is rotated toward y. The `zrot` quantity stored by the class and accessible via `getZRot` and `setZRot` is the negation of the PA.

The methods `alignToEcliptic(), alignToICRS(),` and `alignToInvariable()` are provided to set the y axis to have the specified position angle (in radians, default$= 0$) with respect to the local north directions in the respective coordinate systems. `alignToICRS()`.

4.5.1. Coordinate transformation

Each instance of `Orientation` defines a new 3d coordinate system with new axes. The method `m()` returns a reference to a `Matrix33` that converts a `Vector3` in the ICRS system to a `Vector3` in the newly oriented coordinate system (a rotation matrix). This works, in particular, for unit vectors that represent directions on the celestial sphere. In other words, if `xICRS` is a `Vector3` giving the ICRS direction cosines of a point on the celestial sphere, then `orient.m()*xICRS` will be a `Vector3` giving the direction cosines in the coordinate system described by `orient`.

The methods `fromICRS(), toICRS()` will map 3-vectors from/to ICRS coordinates to/from their equivalents in the system defined by the instance of the `Orientation` class.

4.5.2. Example of projection

Suppose you have an image that is a tangent-plane projection of the sky about some known location `(projRA, projDec)` on the sky, and the projection axes are aligned to local ICRS E and N. You measure the position of a star to be `(xStar, yStar)` (in degrees) in this image and you want to know the ICRS RA and Dec. Here is the code:

```
SphericalICRS pole(projRA, projDec);  // Projection pole: coordinates in radians
Orientation orient(pole);             // Specify orientation: default PA is zero

// Create a Gnomonic instance representing star's location on the sky,
// with constructor including the Orientation of the coordinate system
Gnomonic tp(xStar*DEGREE, yStar*DEGREE, orient);

SphericalICRS icrs(tp);               // Create ICRS coordinates for same location

// Read ICRS RA and Dec into new variables (in radians):
double raStar, decStar;
icrs.getLonLat(raStar, decStar);
```

If you wanted to transform coordinate in the other direction, you create the `icrs` object first, then create `Gnomonic tp(icrs,orient)` to effect the conversion. `tp.getLonLat(xi,eta)` would put the tangent-plane coordinates (in radians) into variables `xi` and `eta`.

5. Cartesian coordinates

Locations in 3d space are represented as instances of the abstract base class `CartesianCoords`. Every instance of the base class represents a definitive point in space, but the coordinate system used depends upon the type of derived class for the instance, in the same way the `SphericalCoords` works for locations on the celestial sphere. The method `getVector()` returns a 3-vector giving the position in the system specified by the derived class. The distance units for `CartesianCoords` are not specified in the code, so the user needs to maintain consistency. All of the orbital-mechanics code will assume units of AU.

???? continue this....