Revised 10 Feb 2013, not up to date yet

Documentation for PixelMap Classes

G. M. Bernstein

Dept. of Physics & Astronomy, University of Pennsylvania

garyb@physics.upenn.edu

1. Dependences

The `PixelMap` classes are placed into the `astrometry` namespace, and make use of the spherical coordinate classes in *Astrometry.h*. As with the *Astrometry.h* classes, linear algebra is assigned to Mike Jarvis's *TMV* package. The typedefs in *Astrometry.h* and *Std.h* provide aliases for the *TMV* classes that are used in `PixelMap` classes: `Vector2` and `Matrix22` are 2-dimensional double-precision vectors/matrices; and `DVector` is an arbitrary-dimension double-precision vector. A few methods are used from `TMV` that would have to be reproduced if another linear algebra package were to be used.

2. `PixelMap`

`PixelMap` is an abstract base clase representing a map from one 2d coordinate space ("pixel" coords) to another ("world" coords). Methods `toWorld()` and `toPix()` execute the forward and inverse maps, respectively. Methods `dWorlddPix` and `dPixdWorld()` return a 2×2 matrix giving the partial derivatives of the forward and inverse maps, respectively, and `pixelArea` returns the world-coordinate area of a unit square in pixel space, *i.e.* returns the (absolute value of the) Jacobian determinant of the forward map at a specified point.

Each `PixelMap` can depend upon a vector of controlling parameters. The current values of the parameter vector are accessed with `setParams()` and `getParams()`. The number of parameters of the map is returned by `nParams()`.

One can call `toWorldDerivs()`, supplying a reference to a $2 \times$ `nparams`() matrix that will be filled with the partial derivatives $\partial[x,y]_{\rm world}/\partial\mathbf{p}$, where $\mathbf{p}$ is the parameter vector, evaluated at the supplied values of $[x,y]_{\rm pix}$. Method `toPixDerivs()` also fills a supplied matrix with the derivatives of the *world* coordinates with respect to parameters (even though one is requesting the pixel coordinates).

There are no constraints on the nature of the "pixel" and "world" coordinate systems, despite the names. No units are assumed. The only quality of the pixel space assumed is that an interval $\Delta[x,y]_{\rm pix} = 1$ is an appropriate step size for calculating numerical derivatives of the map to world coordinates. But you also have the option to change this default pixel-space step size with `setPixelStep()` or read it with `getPixelStep()`. Some implementations may choose to ignore `setPixelStep()` if they have some natural scale in pixel space.

Every `PixelMap` also has a name string that is accessed with `getName()`. The base class has a default such that if no (or null) `name` string is provided, a name `map_NN` will be assigned, with a running index number `NN`. Most derived classes propagate this naming convention.

Other methods that each `PixelMap` implemenation is expected to have, in order to enable complete serialization and deserialization of these classes:

- `duplicate()` returns a `PixelMap*` pointing to a deep copy of itself. An exception is that if you have a `SubMap` or `Wcs` that is created with `shareMaps=true`, then the original and the copy both contain only pointers to component `PixelMaps` that they do not own.
- `write(std::ostream& os)` serializes all information needed to construct the state of the map (except for the name and the pixel step).
- `getType()` returns a short string that identifies the implementation being used, e.g. "`Poly`" for the `PolyMap` class.
- `mapType()` is a static member function returning the same class-identifying string as `getType()`.
- `create(std::istream& is, string name)` is a static member function that will return a pointer to a freshly constructed instance of the class that has deserialized information from the `write()` routine.

3. Implementing a new PixelMap

To derive a functioning class from `PixelMap`, the minimial requirement is to implement the two point-mapping methods `toPix()` and `toWorld()`. A new implementation also requires the `duplicate()` and (de-)serialization routines listed at the end of the previous section. All other `PixelMap` methods have default implementations in the base class.

It would be common for the forward map `toWorld(double xpix, double ypix, double& xworld, double& yworld)` to be defined by some formula for your map. Sometimes the inverse map is easily expressed analytically, but if not, the base class defines the protected method

```
void NewtonInverse(double xworld, double yworld,
                   double& xpix, double& pix,
                   double worldTolerance) const;
```

which can be used to solve for the inverse map `toPix()` by using the known forward map `toWorld()` and its derivative. The solution is done using Newton's iteration: the input values of $\mathbf{x}_p =$

(xpix, ypix) is taken as an initial guess of the inverse solution. The initial guess is mapped to a world point $\hat{\mathbf{x}}_w$ using the forward map, and the iteration follows

$$\mathbf{x}_p \to \mathbf{x}_p + \left(\frac{\partial \mathbf{x}_w}{\partial \mathbf{x}_p}\right)^{-1} (\mathbf{x}_w - \hat{\mathbf{x}}_w). \tag{1}$$

The iteration continues until $|\mathbf{x}_w - \hat{\mathbf{x}}_w|$ is below `worldTolerance` or until more than `PixelMap::NewtonInverse()::` is exceeded (this is coded to 10). Very simple, but unless your starting guess is in a region that is beyond some singularity of the map, it should do well. *Note that it is advantageous to submit a starting* `xpix,ypix` *that was the solution of a neighboring object.* An `AstrometryError` is thrown if the Newton iterations do not converge.

The derivative method `dWorlddPix()` is implemented in the base class by a finite-difference estimate using the `getPixelStep()` value as a step size for the simple numerical derivatives. `dPixdWorld()` is implemented in the base class by taking the matrix inverse of `dWorlddPix()`, and `pixelArea()` is implemented as the determinant of the numerical forward derivatives.

All of the routines related to map parameters are implemented in the base class to have the proper behavior for a map that has *no* free parameters. If your map does have adjustable free parameters, you will have to implement `nParams()`, `setParams()`, `getParams()`, and the `toWorldDerivs()` and `toPixDerivs()` that return derivatives with respect to parameters.

4. Atomic `PixelMaps`

We call a class derived from `PixelMap` "atomic" if it does not result from compounding other `PixelMaps`.

4.1. `IdentityMap`

When you want a map that does nothing. There are no parameters, and the derivatives of the map are identity matrices. The serialization of the `IdentityMap` is empty. `mapType=''Identity''`.

4.2. `ReprojectionMap`

This is a `PixelMap` that embodies any map of the celestial sphere from one coordinate system to another that are both represented by a class derived from `SphericalCoords`. The `ReprojectionMap` is constructed with

```
ReprojectionMap(const SphericalCoords& pixCoords,
                const SphericalCoords& worldCoords,
```

```
                double scale_=1.,
                string name="");
```

The “pix” and “world” coordinate systems are defined by their respective `SphericalCoords` instances. The `PixelMap` is then defined via

$$\begin{aligned} x_{\rm pix} &= {\rm lon}_{\rm pix}/\texttt{scale} & y_{\rm pix} &= {\rm lat}_{\rm pix}/\texttt{scale} \\ x_{\rm world} &= {\rm lon}_{\rm world}/\texttt{scale} & y_{\rm world} &= {\rm lat}_{\rm world}/\texttt{scale} \end{aligned} \tag{2}$$

where the (lon,lat) positions mark the same point on the celestial sphere.

The `ReprojectionMap` class will store duplicates of the two input `SphericalCoord` instances. Be careful if the input coordinate systems have `shareOrient=true.` There are no free parameters.

4.2.1. Example

Suppose you want a `PixelMap` that treats ecliptic coordinates as the “pixel” coordinates and ICRS as the “world” system. And you want the `PixelMap` to work in degree units rather than the radians that are native to the `SphericalCoords` classes. Here is the code:

```
ReprojectionMap map(SphericalEcliptic,
                    SphericalICRS,
                    DEGREE);
double eclipticLon=1.7, eclipticLat=-0.5;  // ecliptic coords in degrees
double icrsRA, icrsDec;     // Want these (in degrees)
// Do a conversion:
map.toWorld(eclipticLon, eclipticLat,
            icrsRA, icrsDec);
```

There are no free parameters in a `ReprojectionMap`. Note that it does not matter what coordinates are stored in the initial `SphericalEcliptic` or `SphericalICRS` used in the constructor: all that matters is the coordinate system that they specify.

4.2.2. Serialized format

The `mapType` is `''Reprojection''`. The serialized `ReprojectionMap` has three lines. The first two are the serialized pixel and world `SphericalCoords` classes. The third line has the scaling factor. The code for (de-)serializing coordinate projections is in *astrometry/SerializeProjection.h, .cpp*. Currently (10 June 2013) the code only knows how to specify ICRS, ecliptic, and gnomonic coordinate systems. These are simply specified by the words “ICRS”, “Ecliptic”, and “Gnomonic,” respectively. The "Gnomonic" specification is followed by the RA and Dec of the projection axis and the position angle (in degrees from N through E) of the y axis of the projected system.

4.3. `PolyMap`

The *PolyMap.h* and *PolyMap.cpp* files declare and define polynomial coordinate maps. They make use of the *utilities2/Poly2d.h* classes. A `PolyMap` is initialized with references to two `Poly2d` instances, defining the two independent functions $x_{\rm world}(x_{\rm pix}, y_{\rm pix})$ and $x_{\rm world}(x_{\rm pix}, y_{\rm pix})$. A third construction parameter is a tolerance, specifying how accurate the solutions for inverse mappings must be. The default value is 0.001/3600 such that a `toWorld()` call will be accurate to 1 milliarcsecond if the units of the world coordinates are degrees. The `setWorldTolerance()` method changes this value.

See the `Poly2d` class documentation for instructions on how to define polynomials of desired order. `PolyMap` makes internal copies of the two `Poly2d` objects at initialization and uses them. These can be viewed with the `get[XY]Poly()` method and are destroyed with the `PolyMap` object. The `setToIdentity()` method sets the coefficients to yield the identity transformation.

The parameters of a `PolyMap` object are the coefficients of the two polynomials (x first, then y). The order of coefficients is defined by `Poly2d`.

`PolyMap::toPix()` uses the `PolyMap::NewtonInverse()` method, and *always* uses the values of `xpix` and `ypix` on input as initial guesses for the Newton method.

4.3.1. Serialized Format

The `mapType` is `''Poly''`. The serialized `PolyMap` has three parts: first come the order and the coefficients of the x polynomial, then those of the y polynomial, then a line containing the world tolerance.

The format of the polynomials is that of the `Poly2d` class in *utilities/Poly2d.h, .cpp.* The first line contains either

- `Sum` $\langle order \rangle$ if the polynomial is constrained to having the sum of x and y orders in a term be $\leq$*order*, or
- `Each` $\langle orderX \rangle \langle orderY \rangle$ if a terms are $x^m y^n$ with $m \leq$*orderX* and $n \leq$*orderY.*

Following lines contain all the necessary coefficients for the polynomial. *utilities/Poly2d.cpp* describes the coefficient order.

4.4. `LinearMap`

Also in *PolyMap.h* is the class `LinearMap`, with transformation defined by the six-element parameter vector $\mathbf{p}$ and the formulae:

$$x_{\rm world} = p_0 + p_1 x_{\rm pix} + p_2 y_{\rm pix} \tag{3}$$

$$y_{\rm world} = p_3 + p_4 x_{\rm pix} + p_5 y_{\rm pix}. \tag{4}$$

The derivatives are all analytic and the `pixelStep` is irrelevant.

4.4.1. Serialized Format

The `mapType` is `''Linear''`. The `LinearMap` is serialized on two lines, the first giving p_0, p_1, and p_2, the second line with p_3, p_4, p_5.

4.5. `TemplateMap1d`

The *TemplateMap.h* file defines `TemplateMap1d`. This is a displacement in either x or in y and is determined by a lookup table of displacements plus a scaling factor that is applied to the displacements found in the lookup table. The scaling factor is the only parameter of the map. A given `TemplateMap1d` either uses the x coordinate as the lookup and produces x displacements; or uses y coordinates as lookup variable and produces a y displacement.

Displacements are specified on a linearly spaced series of nodes. The constructor takes `nodeStart` and `nodeStep` arguments specifying the node positions, and a `deviations` vector of displacements (the length of this vector gives the number of nodes). Values outside the node range use the displacement from the edge node. Displacements are linearly interpolated between nodes.

4.5.1. Serialization format

The `mapType` is `''Template1d''`. The first line of the serialized format is

`[X|Y]` $\langle$*no. nodes*$\rangle$ $\langle$*start node*$\rangle$ $\langle$*node step*$\rangle$.

The second line gives the scaling factor to apply to the displacements.

Third and following lines give the displacement values at the nodes.

5. Composite `PixelMaps`

The classes in this section satisfy the `PixelMap` interface but wrap one or more other `PixelMap` objects to provide more complex behavior.

5.1. `Wcs`

The *Wcs.h* and *Wcs.cpp* files declare and define the `Wcs` class, which extends `PixelMap` by including not only a map from pixel coordinates to world coordinates, but also a `SphericalCoords` instance that defines how the world coordinates map to the celestial sphere.

The `Wcs` constructor takes the following arguments:

- `PixelMap* pm` is a pointer to the map from pixel to world coordinates.
- `const SphericalCoords& nativeCoords` gives the projection in which the world coordinates will be interpreted as lon/lat. A duplicate of the object is created and stored, and is destroyed with the `Wcs`.
- `double wScale` is a scaling factor applied to the world coordinates before the projection interprets them as radians. The default is `DEGREE`, *i.e.* the world coordinates are in degrees.
- `bool shareMap`, if `true`, means that a pointer to `pm` will be stored and it will not be destroyed by `Wcs`. Note this means that parameters of the map are also shared. If `shareMap=false` (the default), a duplicate of `pm` is created and owned by this object.

5.1.1. *Reprojection and* `Wcs` *as a* `PixelMap`

The `Wcs` class implements the `PixelMap` interface. In the simplest case, this is just wrapping the `PixelMap` behavior of the `pm` given on construction of the `Wcs`. However a call to `reprojectTo()` changes this behavior by defining a *target coordinate system* which may differ from the *native* coordinate system. The `toWorld` transformation implemented by the `Wcs` is then defined as follows:

1. The input pixel coordinates $(x_{\rm pix}, y_{\rm pix})$ are transformed to coordinates (lon, lat) by the `PixelMap` that is wrapped by the `Wcs`.
2. The (lon, lat) coordinates are mapped to a location $\mathbf{x}_{\rm sky}$ on the celestial sphere using the native coordinate system supplied at construction of the `Wcs`.
3. The sky location $\mathbf{x}_{\rm sky}$ is mapped to a new pair of coordinates using the projection specified in a call to `reprojectTo()`. The coordinates in this projection are rescaled by `wScale` to yield the "world" coordinates (x_w, y_w).

In other words, when the `Wcs` is accessed through the `PixelMap` interface, it behaves as the original `pm` map followed by a reprojection from the native coordinate system to a new target coordinate system. The free parameters of the `Wcs` are the free parameters of `pm`—the coordinate systems are taken as fixed, with no adjustable parameters.

5.1.2. Methods

The `Wcs` implements all the methods of the `PixelMap` interface. In addition there are these methods specific to `Wcs`:

- `getMap(), getScale(), getNativeCoords()` are accessors to the internal `PixelMap`, the coordinate scaling factor, and the projection to the celestial sphere.
- `reprojectTo(const SphericalCoords& targetCoords)` specifies the second coordinate system used to map the sky position back to 2d coordinates. A duplicate of `targetCoords` is produced and owned by the `Wcs` class.
- `getTargetCoords()` returns (a pointer to) the current target coordinates, if any (else returns zero).
- `useNativeProjection()` discards any specified target coordinate system, so that the `PixelMap` interface returns to using world coordinates in the native projection.

5.1.3. Serialized format

The `MapType` is `"WCS"`. The (de-)serialization routines are not yet implemented for `Wcs` (as of 11 June 2013).

5.2. `SubMap`

The `SubMap` is declared in *astrometry/PixelMapCollection.h* and defined in *astrometry/SubMap.cpp*. A `SubMap` represents the compounded action of zero or more other `PixelMaps`. If the action of the $i^{\rm th}$ map from its pixel to its world coordinates is written as $M_i(\mathbf{x}_{\rm pix})$, then the map from pixel to world coordinates for the `SubMap` is $M_{N-1}(\ldots M_1(M_0(\mathbf{x}_{\rm pix})))$. When constructing a `SubMap` one supplies:

- `const list<PixelMap*>& pixelMaps`, a list of (pointers to) the component pixel maps making up the chain. The component `PixelMaps` can themselves be `SubMaps` or other compound types.

- `string name`, a name for the new map.
- `bool shareMaps`, if set to the default `false`, means that duplicates of all the input component `PixelMaps` will be created and owned by the `SubMap` class. If `shareMaps=true` then pointers to the input instances will be saved, and all parameters will continue to be shared with the input maps. The `SubMap` destructor deletes the component `PixelMaps` only if `shareMaps=false`.

5.2.1. `SubMap` *as a* `PixelMap`

The coordinate mapping methods of `PixelMap` are implemented as expected for chained functions. Note when there are zero maps in the `SubMap` it behaves as the `IdentityMap`. The `getPixelStep(), setPixelStep()` methods access the corresponding routines in the first element of the map chain.

The `nParams(), getParams(), setParams()` methods of `SubMap` work with a parameter vector that is the concatenation of the parameters of all the component maps. Element 0 of the `SubMap` parameter vector is element 0 (if any) of the first `PixelMap` in the `SubMap` chain. Getting or setting parameters of the `SubMap` results in the operation being transmitted to all the relevant component `PixelMaps`.

5.2.2. *Interaction with* `PixelMapCollection`

A `SubMap` can be created as a chain of elements held in a `PixelMapCollection`, as discussed in Section 6. The `SubMap` is aware of where the parameters of its elements live within a master parameter vector for all the maps in the collection. In Section (6) we also discuss a mechanism whereby some elements of the map chain can be "frozen" such that they do not appear in the parameter vectors accessed through the `PixelMap` interface.

5.2.3. *Methods*

The `SubMap` extends the `PixelMap` interface with these methods:

- `const PixelMap* getMap(int i) const` returns a pointer to the component map i. No range checking is done.
- `int nMaps() const` returns the number of component maps.
- `startIndex(int iMap), nSubParams(int iMap)` give the first and number of entries in a global parameter vector corresponding to the parameters from the `iMap`$^{\mathrm{th}}$ map in the chain.

For a `SubMap` created by the user, these index the concatenated vector of the `SubMap`'s component maps. In Section 6 we describe the more complex behavior when a `SubMap` is produced from a `PixelMapCollection`.

- `int mapNumber(int iMap) const` returns the unique id number of a `SubMap` component in the case that the `SubMap` was issued by a `PixelMapCollection`. See 6.

5.2.4. Serialized format

The `MapType` is `"Composite"`. Attempts to serialize a `SubMap` will throw an exception. It is intended to be (de-)serialized only as part of a `PixelMapCollection`.

5.3. CompoundPixelMap

The `CompoundPixelMap` is a **DEPRECATED** predecessor to `SubMap` for the representation of chains of `PixelMap`s. We will not document it further here.

5.4. `TPVMap`

The FITS WCS standard defines maps from pixel to sky coordinates that can be represented by our `Wcs` class wrapping our `SubMap` class. The files *astrometry/TPVMap.h* and *astrometry/TPVMap.cpp* contain functions that allow us to read or write FITS headers that represent the actions of our `Wcs` class.

In practice we do not (yet) attempt to represent the full range of WCS maps sanctioned by the FITS standards. We implement only the pseudo-standard `TPV` map used by Emmanuel Bertin's *SCAMP* program (also sometimes labeled as a `TAN` map). These maps follow a proposal for a FITS WCS standard that was never formally adopted, and has some oddities. But it it widely used. I have implemented a specific subset of the standard that is used by Emmanuel.

5.4.1. The FITS standard

The map from $(x_{\rm pix}, y_{\rm pix})$ to celestial coordinates has three parts in the FITS WCS standard:

1. A linear mapping from pixel coordinates to "intermediate world coordinates" (x_1, y_1) defined by

$$x_1 = \mathtt{CD1\_1}(x_{\rm pix} - \mathtt{CRPIX1}) + \mathtt{CD1\_2}(y_{\rm pix} - \mathtt{CRPIX2}) \quad (5)$$
$$y_1 = \mathtt{CD2\_1}(x_{\rm pix} - \mathtt{CRPIX1}) + \mathtt{CD2\_2}(y_{\rm pix} - \mathtt{CRPIX2}). \quad (6)$$

Quantities in `typewriter font` are FITS keywords. This map can clearly be implemented as a `LinearMap`. The output units are defined by `CRUNIT[12]`, which are string-valued FITS fields that are supposed to have the value `'deg'`. The *TPVMap* code currently assumes this is true, without checking.

2. A polynomial map that transforms the (x_1, y_1) coordinates into the (ξ, η) coordinates in a projection of the celestial sphere. The polynomial definition is as usual:

$$\xi = \sum_{ij} a_{ij} x_1^i y_1^j \quad (7)$$

$$\eta = \sum_{ij} b_{ij} x_1^i y_1^j. \quad (8)$$

The polynomial coefficients are assigned FITS keywords by a quirky convention:

$$\begin{array}{r|l} \texttt{PV1\_0} = a_{00} & \texttt{PV2\_0} = b_{00} \\ \texttt{PV1\_1} = a_{10} & \texttt{PV2\_1} = b_{01} \\ \texttt{PV1\_2} = a_{01} & \texttt{PV2\_2} = b_{10} \\ \texttt{PV1\_4} = a_{20} & \texttt{PV2\_4} = b_{02} \\ \texttt{PV1\_5} = a_{11} & \texttt{PV2\_5} = b_{11} \\ \texttt{PV1\_6} = a_{02} & \texttt{PV2\_6} = b_{20} \\ \texttt{PV1\_7} = a_{30} & \texttt{PV2\_7} = b_{03} \\ \texttt{PV1\_8} = a_{21} & \texttt{PV2\_8} = b_{12} \\ \texttt{PV1\_9} = a_{12} & \texttt{PV2\_9} = b_{21} \\ \texttt{PV1\_10} = a_{03} & \texttt{PV2\_10} = b_{30} \\ \texttt{PV1\_12} = a_{40} & \texttt{PV2\_12} = b_{04} \\ \texttt{PV1\_13} = a_{31} & \texttt{PV2\_13} = b_{13} \\ \texttt{PV1\_14} = a_{22} & \texttt{PV2\_14} = b_{22} \\ \texttt{PV1\_15} = a_{13} & \texttt{PV2\_14} = b_{31} \\ \texttt{PV1\_16} = a_{04} & \texttt{PV2\_14} = b_{40} \end{array} \quad (9)$$

Note there are no `PV[12]_3` or `PV[12]_11` terms (according to the convention they are meant to be coefficients for radial r and r^3 terms, which are not analytic at the origin and hence not useful to us.) The FITS convention is that any missing coefficient is zero, hence the order of the polynomial is determined by the largest `PV`x`_`y that is present in the FITS header.

3. A deprojection from the (ξ, η) coordinates onto the celestial sphere. Many projections are in principle possible and specified by the `CTYPE[12]` keywords, but SCAMP always uses the gnomonic projection that is declared by setting `CTYPE1=RA---TAN` and `CTYPE2=DEC--TAN`, or `RA---TPV` and `DEC--TPV`. Any other values for these keywords throw an `AstrometryError`. The projection pole RA and Dec in the ICRS system are given as degree values in the fields `CRVAL1` and `CRVAL2`, respectively. The gnomonic projection is assumed to have its η axis pointing along the north ICRS meridian, *i.e.* position angle zero.

5.4.2. *Implementing* `TPV` *as a* `Wcs`

The FITS `TPV` standard is implemented as a `Wcs` having its `nativeCoords` equal to a `Gnomonic` projection oriented to the ICRS meridian at the projection pole. The `Wcs` wraps a `SubMap` consisting of a `LinearMap` followed by a `PolyMap`. If no `PV` terms are found in the header, the polynomial map is omitted and the linear map is used without it.

We declare three functions that can be used to convert between FITS-standard headers and our `astrometry` classes.

- `Wcs* readTPV(const img::Header& h, string name="")` can read a FITS header (in the `img::Header` class), extract all the keywords described above to specify a WCS, and return a pointer to a new `Wcs` object implementing this transformation. An optional name is assigned to the `Wcs`. It component `SubMap` has the same name.

- `img::Header writeTPV(const Wcs& w)` writes a FITS header meeting the TPV pseudo-standard. This will throw an exception if the input `Wcs` is not in the form of a `LinearMap` and/or `PolyMap` in sequence.

- `Wcs* fitTPV` is a function that will fit a TPV-form WCS to approximate the behavior of an arbitrary `Wcs` provided as input. The full declaration is

  ```
  Wcs* fitTPV(Bounds<double> b,
              const Wcs& wcsIn,
      const SphericalCoords& tpvPole,
      string name="",
      double tolerance=0.0001*ARCSEC/DEGREE);
  ```

  The polynomial coefficients of the output TPV-format map are solved to minimize the RMS deviation from the `WcsIn` over the rectangular region `b` (see the *utilities2/Bounds.h* file for info on this class). The polynomial order is increased until this RMS deviation is < `tolerance`. There are `startOrder` and `maxOrder` constants defined in *TPVMap.cpp*, currently 3 and 5, respectively. If `maxOrder` is exceeded, an `AstrometryError` is thrown. Note that the usual convention for FITS WCS systems is to express world coordinates in degrees, so the default `tolerance` is 0.1 milliarcsec. The output `Wcs` is defined to have its projection be `Gnomonic` about the `tpvPole` position.

6. `PixelMapCollection & SubMap`

When reconciling world-coordinate maps for a set of data / reference catalogs, it is typical to have a large number of "building block" coordinate maps that are put together in different combinations to maps parts of individual exposures. `PixelMapCollection` (PMC) is a class that serves as a warehouse for all these building blocks, puts them together into any specified chain to form the complete WCS transformations, and facilitates bookkeeping of the parameters of these building blocks within a global parameter vector during a fitting process. The `SubMap` can wrap any chain of `PixelMaps` from a PMC and keep track of where their parameters live within the global parameter vector. The `PixelMapCollection` also controls the creation and destruction, serialization and de-serialization of a full complement of `PixelMap` and `Wcs` components needed to describe a set of image.

6.1. Concepts

A PMC consists of the following kinds of objects:

- *Atomic maps:* these are irreducible `PixelMaps`. Each element of the PMC must have a unique name, by which it is accessed. The PMC maintains a global parameter vector that is the concatentation of all the parameters of its atomic maps.
- *Chains:* A compounded sequence of atomic maps can be assigned to a named chain. The name of the chain must not duplicate the name of any other chain or atomic map. The parameters of a chain are the union of its member atomic maps' parameters.
- *WCSs:* A map (either atomic or chain) associated with a projection of world coordinates back to the sky. Duplication of WCS names is not allowed, but a WCS can have the same name as an atomic or chain map.

The PMC can *issue* a pointer to a `SubMap` that wraps any atomic map or chain in the collection—you request this by the name of the map or chain that you want. You can use this `SubMap` for mapping and fitting coordinates. The `SubMap` knows where the parameters of its constituent maps live within the PMC's global parameter vector. The PMC keeps track of all the `SubMaps` it has issued, keeps their parameter indices up to date, and deletes them upon destruction of the PMC.

You can tell the PMC to fix the parameters of any atomic map (or chain of them) to their current values. These will then no longer appear as free parameters in the `PixelMap` interfaces to the `SubMaps`. You can later free these parameters.

You can also *issue* a pointer to a `Wcs` object that realizes any of the WCS systems that the PMC knows about. Again, the request is by name, and the PMC keeps track of and deletes all the `Wcs`'s it issues.

In addition to *issuing* a realization of any map or WCS, the PMC can *clone* them, producing for you a pointer to a new `SubMap` or `Wcs` object. The difference is that a clone is a fresh deep copy, decoupled from the global parameter vector of the PMC. Unlike issued objects, these remain valid after the PMC itself is deleted.

At any point the definitions and current parameter values of all these elements can be serialized to a stream. (Optionally you can serialize only those elements needed to build a particular map or WCS).

There are multiple ways to add new maps, chains, and WCS's to the PMC. The PMC can *learn* an existing map, essentially creating and storing its own duplicate that you can access by the name of the original object. Composite maps (*e.g.* `SubMaps`) can be learned as well—they define a new chain, and their atomic elements are learned. The PMC can also learn a WCS by being handed an existing `Wcs` object. A PMC can also learn the entire contents of another PMC.

Whenever a PMC is learning about some existing object, there is a flag `duplicateNamesAreExceptions` for the operation which determines the action taken if the object to be learned has a name that duplicates a name already in the PMC. If this flag is `false` (the default), the new object is ignored and we assume that the previous object of the same name can be used in its place. This is the desired situation when we are for example learning the WCS systems of many exposures that share common distortion maps for a given CCD. If the flag is `true`, then duplicate names throw exceptions.

A chain can also be *defined* by specifying the chain of maps that it is made of. A new WCS can be defined from giving the name of its coordinate map and supplying a `SphericalCoords` object defining the projection onto the sky.

The third way to enter new maps or WCS's into the PMC is by deserializing from a stream.

6.2. Methods

6.2.1. Building the collection

The constructor for `PixelMapCollection` creates an empty collection. The following methods expand the collection (see also the serialization methods below). Note that all of the `learn` methods take a `duplicateNamesAreExceptions` flag as described above.

- `learn(PixelMapCollection& source)`: duplicate the names, characteristics, and current parameters of everything in the `source` collection.
- `learnMap(const PixelMap& pm)`: duplicate the name, characteristics, and current parameters of `pm` into this collection. Note that if `pm` is a `SubMap`, all its component maps are learned as well. If `pm` is a `Wcs`, it is learned as a new chain including a `ReprojectionMap`, not as a new WCS.
- `learnWcs(const Wcs& pm)`: duplicate the name and projection of `wcs` and learn its underlying `PixelMap` as well.

- `defineChain(string chainName, const list<string>& elements)`: Define a new map chain with the given name and compounding the maps named by the strings in `elements`.
- `defineWcs(string wcsName, const SphericalCoords& nativeCoords, string mapName, double wScale = DEGREE)`: define a WCS system to be the map described by `mapName` followed by projection to the sky described by the `nativeCoords` system.

6.2.2. Extracting collection elements

These methods produce pointers to `SubMap`s or `Wcs`'s realizing the maps and WCS's known to the collection. The `issue` methods give pointers to objects maintained by this `PixelMapCollection` and linked to its global parameter vector. The `clone` methods return pointers to deep copies that are decoupled from this PMC. All throw exceptions if an unknown name is requested.

- `SubMap* issueMap(string mapName)`
- `Wcs* issueWcs(string wcsName)`
- `PixelMap* cloneMap(string mapName)`
- `Wcs* cloneWcs(string wcsName)`

6.2.3. Parameter manipulation and bookkeeping

- `void setParams(const DVector& p)`: set the global parameter vector from `p`.
- `DVector getParams()`: return the global parameter vector.
- `int nParams()`: return the size of the global parameter vector.
- `void setFixed(list<string> nameList, bool isFixed)`: Declare that the parameters of all the maps with names in `nameList` are to be fixed (freed) if `isFixed=true` (`false`). If a map's parameters are fixed, then they are removed from the global parameter vector. Any issued `SubMap`s that refer to this map are updated to indicate appropriately smaller number of free parameters and index reference for all parameters are updated. If any of the names is of a chain, then all members of the chain have their parameters fixed (or freed).
- `void setFixed(string name, bool isFixed)`: same as above, just freeing/fixing a single map.
- `bool getFixed(string name)`: report whether parameters of the map with `name` are free or fixed. Note the potential for confusion about chains, which can think they are fixed but have had some of their elements freed by other operations.

- `bool mapExists(string name)`: reports whether there is an atomic map or a chain with this `name`.
- `bool wcsExists(string name)`: reports whether there is a WCS with this `name`.
- `int nWcs()`: report number of WCS's known to this collection.
- `int nMaps()`: report number of maps known to this collection (atomic maps plus defined chains).
- `int nAtomicMaps()`: report number of atomic maps known to this collection.
- `int nFreeMaps()`: report number of atomic maps known to this collection that have free parameters.

6.2.4. Serialization

All of the (de-)serialization of `PixelMaps` and WCS's is intended to be done via the `PixelMapCollection` methods. *The class must be told of the existence of every kind of* `PixelMap` *that it will encounter in deserialization.* If you will be deserializing any atomic map type beyond the `Identity, Reprojection, Poly,` and `Linear` maps, you need to put this in your code once.

```
PixelMapCollection::registerMapType<PolyMap>()
```

This templated function has told the `PixelMapCollection` class to look for serialized `PolyMaps` and given it the address of the function for deserializing them.

The methods for (de-)serialization are:

- `void write(ostream& os)`: serializes the entire collection to the stream.
- `writeMap(ostream& os, string name)`: serialize just the named map and anything it depends upon.
- `writeWcs(ostream& os, string name)`: serialize just the named WCS and anything it depends upon.
- `bool read(istream& is, string namePrefix="")`: deserialize all the maps and WCS's from the input stream and add them to the collection. Returns `true` on success, `false` if the stream is not a serialized `PixelMapCollection`, and throws an exception for format errors.

6.2.5. Serialized format

A serialized `PIxelMapCollection` has the following format. Any blank line or any line starting with `#` is ignored as a comment.

1. The stream must start with a single line starting with the word `PixelMapCollection`.
2. Each atomic map is specified by this sequence:
   (a) A line containing ⟨*mapType*⟩ ⟨*map name*⟩ ⟨*pixel Step*⟩.
   (b) Further lines with the serialized content produced by `PixelMap.write()`.
3. Each chain is specified by this sequence:
   (a) A line containing `Composite` ⟨*map name*⟩.
   (b) Further lines with the names of the elements of the chain, any number per line.
4. Each WCS is specified by this sequence:
   (a) A line containing `WCS` ⟨*WCS name*⟩ ⟨*pixel map name*⟩.
   (b) A line with the serialized version of the native coordinate system.
   (c) A line with the coordinate scaling factor `wScale`.