CVS $Revision: 1.4 $Date: 2011/04/13 03:36:23

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()`. **Note that the size of the parameter vector returned by `getParams()` is allowed to exceed the number of parameters, so you must consult `nparams()` to know how many parameters there are.** (In particular if there are no parameters, `getParams()` might return a 1-dimensional vector since some linear algebra routines might not like having zero-dimensional vectors.)

When calling `toWorld()` one can optionally supply 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 `toPix()` can also be asked to supply partial derivatives of the inverse map with respect to the map parameters.

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 $\Delta[x, y]_{\mathrm{pix}} = 1$ 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()`.

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()`. 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 solution of 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 $\mathbf{x}_p = (\mathtt{xpix}, \mathtt{ypix})$ 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

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

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 versions of `toWorld()` and `toPix()` that return derivatives with respect to parameters.

4. Derived 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.

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(SphericalCoords* pixCoords,
                SphericalCoords* worldCoords,
                double scale_=1.);
```

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

$$\begin{array}{ll} x_{\rm pix} = {\rm lon}_{\rm pix}/\texttt{scale} & y_{\rm pix} = {\rm lat}_{\rm pix}/\texttt{scale} \\ x_{\rm world} = {\rm lon}_{\rm world}/\texttt{scale} & y_{\rm world} = {\rm lat}_{\rm world}/\texttt{scale} \end{array} \quad (2)$$

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

The ReprojectionMap class will assume ownership of the two SphericalCoords objects that it points to. It will reset their lon/lat positions in order to effect pixel maps **and it will destroy them when the ReprojectionMap is destroyed. So you point to "spare" copies during construction.**

Note also that if one of the SphericalCoords is a TangentPlane or SphericalCustom object, it will contain a pointer to an Orientation object. You must not alter or destroy that object as long as the ReprojectionMap is being used, and you are still responsible for destroying the Orientation.

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(new SphericalEcliptic,
```

```
                    new 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);
// the Spherical* instances are destroyed when map is destroyed.
```

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.

4.3. CompoundPixelMap

A `CompoundPixelMap` is the chained application of a sequence of N `PixelMaps`. Let map i treat (x_{i-1}, y_{i-1}) as its "pixel" coordinates and (x_i, y_i) as its "world" coordinates. Then (x_0, y_0) are the "pixel" coordinates of the `CompoundPixelMap` and (x_N, y_N) are the "world" coordinates of the `CompoundPixelMap`. The `toWorld()` method of `CompoundPixelMap` simply applies the `toPix()` methods of all component maps in order. `CompoundPixelMap.toWorld()` applies its consituent `toWorld()` maps in *reverse* order. The overall derivative matrix of the `CompoundPixelMap` is the product of its constituents'.

4.3.1. Building a `CompoundPixelMap`

You construct `CompoundPixelMap` with no arguments or with a pointer to a single `PixelMap`. The `append()` and `prepend()` methods take pointers to another `PixelMap` that will be added to the beginning or end of the transformation list. The `PixelMap` at the front of the list is the one applied first in the `toWorld()` mapping. `CompoundPixelMap.setPixelStep()` calls the `setPixelStep()` method of the `PixelMap` at the front of its list.

`CompoundPixelMap` **only saves a pointer to each of its component maps. The user is responsible for keeping the component maps in existence until the** `CompoundPixelMap` **is no longer in use and for destroying the components thereafter.**

4.3.2. Parameters

The parameter vector of a `CompoundPixelMap` is the concatenation of the parameter vectors of all of its constituent `PixelMaps`, in the order that they appear in the list. The `get/setParams()`

method will read/write parameters to/from all the constituent `PixelMaps`. Note that this will have the side effect of altering the constituent maps, even for use outside of this `CompoundPixelMap` context.

The `CompoundPixelMap` class knows how to propagate derivatives with respect to parameters of every transformation in its chain. A `CompoundPixelMap` can be used whenever any other kind of `PixelMap` is valid.

5. `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 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 pixel coordinates that solved the previous call as the initial guess for the next call.

6. `LinearMap`

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

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

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

7. `SCAMPMap`

The files *shapes/SCAMPMap.h* and *shapes/SCAMPMap.cpp* derive from `CompoundPixelMap` a new class, `SCAMPMap`, that implements the pseudo-standard world coordinate system (WCS) maps used by Emmanuel Bertin's *SCAMP* program. These maps follow a proposal for a FITS WCS standard that was never formally adopted, and has some oddities. But it it widely used and can be specified by a series of FITS keywords. I have implemented a specific subset of the standard that is used by Emmanuel.

7.1. 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

$$x_1 = \mathtt{CD1\_1}(x_{\rm pix} - \mathtt{CRPIX1}) + \mathtt{CD1\_2}(y_{\rm pix} - \mathtt{CRPIX2}) \tag{5}$$

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

   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 `SCAMPMap` 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:

$$\xi = \sum_{ij} a_{ij} x_1^i y_1^j \tag{7}$$

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

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

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

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 PVx_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`. Any other values for these keywords throws 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.

7.2. `SCAMPMap` implementation

The FITS WCS map standard is implemented by making `SCAMPMap` derive from `CompoundPixelMap`. The basic `SCAMPMap` is defined as the map from $(x_{\rm pix}, y_{\rm pix})$ (in pixel units) to (ξ, η) (in **degrees**), with the latter coordinates taken to be in the gnomonic projection about the `CRVAL[12]` pole specified in the FITS header. Internally, this is a `LinearMap` followed by a `PolyMap` that encode steps (1) and (2) of the WCS definition above. These maps are created, stored, and destroyed by the `SCAMPMap` class. The user may obtain `const` pointers to these with the methods `linear()` and `poly()`. The `Orientation` of the gnomonic projection specified by the FITS header is also constructed, stored, and destroyed by the `SCAMPMap` class. A reference to it can be obtained by the `orientFITS()` method.

If no `PV` terms are found in the header, the polynomial map is omitted and the linear map is used without it.

Optionally the `SCAMPMap` can create a `ReprojectionMap` and append it to the `PixelMap` chain so that the world coordinates (ξ, η) are projected into gnomonic project specified by an `Orientation` of the user's choosing. This is useful, for instance, if we want to map several exposures' pixels into a common tangent-plane system. A reference to the `Orientation` in which the `SCAMPMap` world coordinate system is defined can be obtained from the `projection()` method.

7.3. Constructing a `SCAMPMap`

The constructor argument is a reference to an `image::ImageHeader` that must define all of the keywords needed to specify the standard map as described above. The linear map and the orientation of the WCS gnomonic projection are read using the `ReadCD()` function in *SCAMPMap.cpp*. Another function, `ReadPV()`, is used to read the ξ and η polynomial coefficients from an `ImageHeader` and produce `Poly2d` instances from them.

The constructor will optionally take a pointer to an `Orientation` that will be used to define the `TangentPlane` coordinate system of the output coordinates. If a zero pointer is passed (the default), then the `Orientation` of the WCS system is used and the `SCAMPMap` world coordinates will coincide with the WCS defined by the keywords. If a different `Orientation` is passed, then it is copied by the `SCAMPMap`, and the user does *not* have to maintain the object whose pointer was given during construction.

In short: if you construct a `SCAMPMap` from an `ImageHeader`, you have a completely self-contained instance of a `PixelMap` with no free parameters, optionally mapping into any gnomonic projection of the sky that you choose. All of the elements of the map are stored inside the class and appropriately cleaned up upon its destruction.

7.4. Fitting a FITS-style WCS map to a `PixelMap`

SCAMPMap.h declares a function `FitSCAMP()` that creates a SCAMP-style FITS WCS map that is a close fit to the action of any `PixelMap` that you give as input. The purpose is to create a header that can be installed in a FITS image that will closely approximate the behavior of any `PixelMap` that you have determined to be a good astrometric solution for your image. The function declaration is

```
img::ImageHeader
astrometry::FitSCAMP(Bounds<double> b,
                     const PixelMap& pm,
                     const Orientation& pmOrient,
```

```
const SphericalCoords& pole,
double tolerance=0.0001*ARCSEC/DEGREE);
```

The map M returns the $(\xi,\eta)_{\rm SCAMP}$, in degrees, of a gnomonic projection centered at the designate `pole`. **The input `PixelMap pm` is assumed to be a map from pixel coordinates to a degree-valued $(\xi,\eta)_{\rm pm}$ coordinate in a `TangentPlane` (gnomonic** projection with `Orientation pmOrient`. The function finds the polynomial coefficients of a SCAMP-style map that brings $(\xi,\eta)_{\rm SCAMP}$ to the same point on the celestial sphere as $(\xi,\eta)_{\rm pm}$ for any pixel coordinate inside the rectangular region described by the `Bounds` object `b` (see the *utilities2/Bounds.h* file for info on this class).

The polynomial coefficients of the output `SCAMPMap` are solved to minimize the RMS deviation between the map M and the map `pm` over the rectangular region `b`. The polynomial order is increased until this RMS deviation is $<$ `tolerance`. There are `startOrder` and `maxOrder` constants defined in *SCAMPMap.cpp*, currently 3 and 5, respectively. 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 function output is an `ImageHeader` that contains the keywords defining a map conforming to the FITS WCS pseudo-standard and readable by *SCAMP*, *DS9*, and other common code. The FITS WCS coordinates will be defined in a gnomonic projection about `pole` supplied to the function. This means that `CRVAL[12]` will be set to the ICRS RA/Dec of this pole. The `CRPIX[12]` reference pixel value and the `CD[12]_[12]` matrix elements will be selected so that the linear part of the WCS map matches the full map to first order at the center of the `Bounds`.

7.4.1. Implementation

A rectangular grid of $\approx$ `nGridPoints` (currently set to 400) is placed spanning the given pixel-space bounds. Each is mapped to the celestial sphere with `pm` and then remapped into the projection desired for the FITS-style WCS map. A linear least-squares adjustment to all the polynomial mapping coordinates is made. If the resulting map is not of sufficient accuracy, the WCS polynomial order is increased, and we try again. If `maxOrder` is exceeded, an `AstrometryError` is thrown.

8. `MapCollection` & `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. `MapCollection` 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. `SubMap` is derived from `PixelMap` and wraps any PixelMap by adding information about where the parameters of that particular PixelMap live within the global parameter vector.

The methods for MapCollection are:

- The constructor simply makes an empty collection. Copy constructor and assignment are hidden to avoid ownership ambiguities for the PixelMaps in the collection.
- add(PixelMap* pm) will add pm to the internal list of PixelMaps available to serve as building blocks. The MapCollection class assumes ownership of the PixelMap pointed to by pm and will delete it when the MapCollection is destroyed. The add returns a MapIndex object which henceforth can serve to retrieve that PixelMap from the collection.
- nMaps() returns the number of PixelMaps that have been put in the collection so far.
- nparams(), setParams(), getParams() allow access to/from a vector of parameters that is the union of parameters from all maps that have been placed into the collection.
- MapChain is a public subclass that is a sequence of MapIndex values. A MapChain hence represents an ordered sequence of transformations. MapChain has these methods:
  - append(MapIndex m), append(MapChain& mc) add element(s) to the back end of the chain. When created, a MapChain is empty.
  - The usual container-class methods front(), size(), empty(), begin(), end() and iterators are available. Currently, MapChain is implemented as a list, so these methods are just inherited.
- issue(MapChain& chain) returns a pointer to a SubMap that wraps a CompoundPixelMap implementing a coordinate transformation that is composed of the sequence of building blocks requested in the chain. The MapCollection class keeps track of all the CompoundPIxelMap and SubMap classes it has created, and deletes them upon destruction of the parent MapCollection, so you can use them without worrying about cleaning up.

SubMap is derived from PixelMap. A SubMap is initialized with a pointer to another PixelMap and the SubMap's transformations and parameters all refer to those of this PixelMap. **Note that the SubMap does not own its parent PixelMap, so the user must insure that the PixelMap is maintained during use and destroyed on completion. If you get a SubMap pointer from the MapCollection:issue() method, the destruction is done for you by MapCollection. SubMap.setParams() alters the parameter set of the PixelMap.**

SubMap extends the PixelMap interface only by adding two public vector<int> members, startIndices and nSubParams, which tell you where this SubMap's parameters live within a global parameter vector. For example if startIndices is {7,22} and nSubParams is {3,4}, it means that the first three parameters of this `SubMap` are indexed as 7–9 in the global vector, and the next 4 parameters of the `SubMap` are stored in positions 22–25 of the global map.

It is up to the user to fill up and use these two vectors. However a `SubMap` returned by the `MapCollection::issue()` method will have them properly set up.