The Geometry Package for the Pierre Auger Observatory

Lukas Nellen*, I de Ciencias Nucleares, UNAM, 04510 Mexico, D.F.
Stefano Argiro, University of Torino, Italy
Thomas Paul, Northeastern University, Boston
Troy Porter, Louisiana State University, Baton Rouge
Luis Prado Jr, State University of Campinas, Campinas

Abstract

The Pierre Auger Observatory (PAO) [1] calls for two sites with multiple semi-autonomous detection systems. For example, the site currently under construction comprises 24 fluorescence telescopes pointing in different directions and 1600 surface particle detectors spaces 1.5 km apart. Each component, and in some cases each event, provides a preferred coordinate system for simulation and analysis. To avoid a proliferation of coordinate systems in the offline software [2] of the PAO, we have developed a geometry package, implemented in C++, that allows the treatment of fundamental geometrical objects in a coordinate-independent way. This package makes transformations between coordinate systems transparent to the user.

The geometry package allows easy combination of the results from different sub-detectors, at the same time as ensuring that effects like the Earth's curvature, which is non-negligible on the scale of a single Auger site, are dealt with properly.

The internal representations used are Cartesian. For interfacing, including I/O, the package includes support for Cartesian coordinates, geodetic (latitude/longitude and UTM), and astrophysical coordinate systems.

INTRODUCTION

The geometry requirements of the Pierre Auger Observatory is different from that of a typical high energy physics experiment. First, the sheer size of the detector is much larger, each site measuring ≈ 40 km in diameter. Over this scale, details like the earth curvature are not negligible. Second, in a cosmic ray experiment, there is no single, natural coordinate system. Instead, each component of the detector (e.g., fluorescence eye) and/or event (e.g., the air shower itself) has a preferred coordinate system. Tracking many coordinate systems in general without special support can be tedious and error prone. Both problems can be alleviated by having each geometry object (e.g., point and vector) keep track of the coordinate system in which it is represented. A useful consequence of this is that it becomes possible to operate on objects in a coordinate-independent way.

We implement these ideas in C++ as part of the geometry package of the Offline Software [2] of the PAO.

Besides the general abstraction of affine linear algebra, the geometry package for the PAO also includes support for geodetic and astronomical coordinate systems. The package also adds creator functions for coordinate systems following the conventions of the PAO [3].

* lukas@nuclecu.unam.mx

AFFINE LINEAR ALGEBRA

Most implementations of (affine) linear algebra [5, 4, 6, 7] implement objects like points, vectors, and matrices as collections of coordinates together with a set of operations on them. This means that using such a package relies on a convention to define which coordinate system a given object's coordinates are specified in. This is not a problem if there is a single coordinate system used in an application. Otherwise, the user has to code coordinate transformations at the required places. Particularly, in a modularised framework, one would typically rely on the use of a standard coordinate system when passing information across module boundaries. This introduces unnecessary transformations if several modules that use the same coordinate system are executed in sequence. And even in the general case, one ends up transforming coordinates twice when crossing a module boundary: first from the local coordinate convention of one module into the reference coordinate system, and then from the reference system into the preferred system for the second module.

The alternative, implemented in the geometry package presented here, is to have each object keep track of the coordinate system used for the current representation of the object. This way, we leverage the advantage of the higher level of abstraction possible in object oriented programming and can write simply

```
Vector v1, v2;
...
Vector v3 = v1 + v2;
```

without worrying about the coordinate systems used to represent the vectors internally.

Operator overloading in C++ allows us to maintain a familiar notation for the operations in the geometry package. This improves the readability of the code. The strong typing of C++ helps to detect attempts to perform undefined operations, e.g., summing of two points, at compile-time.

When creating an object like a vector, or when the user wants to extract the coordinates, e.g., for passing them to an existing plotting package, one has to specify the coordinate system in which the coordinates are defined:

```
CoordinateSystemPtr cs;

Vector v(1, 2, 3, cs);

double x = v.GetX(cs);

boost::tie(x, y, z) = v.GetCoordinates(cs);
```

Advantages of abstraction

The advantages of using coordinate system independent geometry are that user is not required to track coordinate systems, nor required to rely upon a global convention. Geometry objects can be passed along module boundaries without any transformation or common convention.

Since the penalty for using a specialised coordinate system at a given point in the processing is reduced, code can become more readable since the best coordinate system for a given task can be used. For example, when importing geometry information from external, legacy software, one can define a coordinate system following the conventions of the external software and create geometry objects directly from the coordinates provided. To set the position of a particle generated in an air shower simulated by the CORSIKA [8] Monte Carlo, one simply writes:

```
Point pos(corsikaParticle->fX*cm,
          corsikaParticle->fY*cm,
          0.*cm,
          GetCorsikaCoordinateSystem());
Vector  p(corsikaParticle->fPx*GeV,
          corsikaParticle->fPy*GeV,
          -corsikaParticle->fPz*GeV,
          GetCorsikaCoordinateSystem());
```

without having to code the explicit coordinate transformation separately for position and direction of motion.

Risks of abstraction

Since the internal representation of an object can be in an arbitrary coordinate system, the geometry package potentially has to transform representations to a common representation before operating on the internal representation. If this happens in an uncontrolled way, it can lead to unnecessary transformation. This can affect both the runtime performance and the numerical precision of a program.

When (not) to transform

In order to avoid unnecessary transformation, the geometry package makes the following *guarantee*:

> If two objects are represented internally in the same coordinate system, no transformation will occur and the result will be represented in the same coordinate system.

To help code performance to benefit from this guarantee, all objects provide a `TransformTo` member function that forces the internal representation to use the specified coordinate system.

In practice, a module that uses a geometry objects created in a different module will force the internal representation into its preferred coordinate system, thereby avoiding uncontrolled transformations further down. Compared to the standard scenarios outlined above, where each object gets transformed once on exit of a module and then on entry to the next, we still save one coordinate transformation since both transformations occur at once.

Implementation Issues

Coordinate system independent geometry adds the overhead of first verifying if two objects are represented in the same coordinate system and, if the coordinate systems differ, carrying out an additional coordinate transformation. As explained in the previous section, the client code has the possibility to force the representation of all objects into a common coordinate system, thereby avoiding further coordinate transformations. To minimise the remaining, unavoidable penalty of comparing coordinate systems, such comparisons are implemented as object comparisons. This means two objects are considered to be in the same coordinate system only if the coordinate system objects they are defined in coincide. It is not sufficient that two objects define the same coordinate system. Client code therefore has to keep a handle to a coordinate system once it is created (e.g., the local coordinate system at a point on the Earth's surface) and not re-create the coordinate system on every use.

As an optimisation of coordinate transformation, a caching mechanism stores recently used transformation matrices between coordinate system. This way, most coordinate transformations can be reduced the the minimum number of required operations.

The lifetime of a coordinate system is determined by the life-time of all handles to it, which can be explicitly held in client code and implicitly held in objects defined using it. The moment the last handle to a coordinate system is deleted the coordinate system object itself has to be deleted. This is a typical application for a shared pointer, since we have no control over the lifetime of all objects that use a coordinate system. We provide a `CoordinateSystemPtr` based on the BOOST smart pointer library.

COORDINATE SYSTEM REGISTRIES

In order to define a coordinate system, one has to refer to an existing coordinate system. Eventually, this leads to a chicken-and-egg problem, which is resolved by requiring the existence of an ultimate root-coordinate system. This coordinate system has to be fixed ultimately by convention. In order to avoid the reliance on the existence of this ultimate coordinate system, whose definition is irrelevant for the client code, the geometry package for the PAO provides a registry of a few well-known coordinate systems: an earth centred, earth fixed system, and a site local system (according to conventions). Other, specialised coordinate systems are available from different parts of the detector or event data structures. Thus, the user is mostly isolated from the creation of coordinate systems and the intended abstraction is maintained.

GEODETIC COORDINATES

Position information of the detector components (fluorescence eyes, surface detector stations) is provided by the surveyors in Universal Transverse Mercator (UTM) coordinates. The UTM projection is a conformal map on a reference ellipsoid, ideal for navigation. However, it defines a non-linear coordinate system which is difficult to visualise. In order to define a UTM transformation to a latitude/longitude pair, one needs the reference ellipsoid used to select a concrete member of the continuous family of transformations defined by the general UTM prescription.

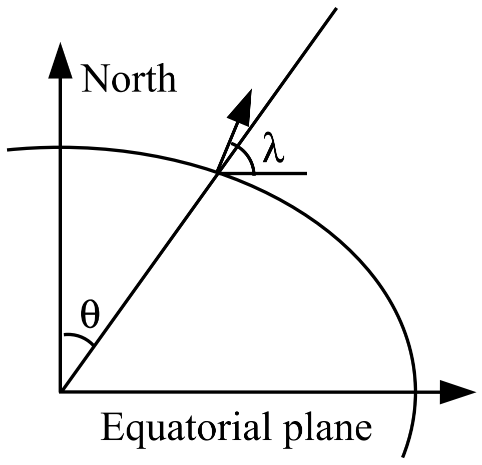

Figure 1: The geodetic latitude λ is defined as the angle between the local vertical and a plane parallel to the equatorial plane. For an elliptical shape, it is not just the comlement of the zenith angle θ in the definition of spherical coordinates.

The definition of latitude in geodesy uses the local vertical as defined by a reference ellipsoid, not just the angle $\lambda = 90° - \theta$, where θ is usual zenith angle in spherical coordinates (fig. 1).

To support the transformations to and from geodesic coordinates, the geometry package provides the `UTMPoint` class for handling all the transformations between Cartesian and UTM coordinates of a point on the Earth.

QUALITY ASSURANCE

The geometry package has been developed in parallel with a comprehensive set of unit tests. The tests are integrated into the general test suite of the offline framework of the PAO.

The unit tests are developed in parallel with the geometry package, thereby providing rigorous testing during every single step of the development, enormously reducing the number of bugs to be resolved later on. It also allowed us to do some major refactoring of the code without breaking any existing code while at the same time making further developments significantly simpler.

EXPERIENCES

The experiences with the geometry package were very positive. The integration of data from different detector components (surface and/or one or more fluorescence stations) was significantly simplified due to the coordinate system independence. So far, we have not noticed any performance penalties due to the additional tracking of a coordinate systems associated with each object.

FUTURE DEVELOPMENTS

For the immediate future, we plan to add astronomical coordinate systems to the package. We are also working on extending the affine geometry part by adding line and plane objects.

An extension still under investigation is error propagation together with coordinate system independent geometry. It requires handling and transforming an error matrix for geometry objects, and an interface for obtaining the information in a reasonable way that does not give up the coordinate independence of the package. Error propagation is important when using the geometry package in reconstruction. For example, when we obtain the axis of a shower from the intersection of two shower-detector planes, each known only with a certain error, we need to obtain error estimates on position and direction of the shower axis.

ACKNOWLEDGEMENTS

We would like to thank all our collaborators, particularly our patient testers, for their help during the development of the Offline Framework and the geometry package.

REFERENCES

[1] Pierre Auger Observatory homepage, http://www.auger.org/

[2] The Offline Framework of the Pierre Auger Observatory, L. Nellen et. al., these proceedings;

[3] Conventions of the Pierre Auger Observatory, https://edms.cern.ch/cedar/plsql/doc.info?document_id=317390

[4] BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/

[5] Blitz++, http://www.oonumerics.org/blitz/

[6] CLHEP, http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/

[7] ROOT, http://root.cern.ch/

[8] D. Heck, J. Knapp, J.N. Capdevielle, G. Schatz, T. Thouw, Report FZKA 6019 (1998); http://www-ik.fzk.de/˜heck/corsika/