RecPack, a general reconstruction toolkit

A. Cervera-Villanueva* and J.J. Gómez-Cadenas
IFIC, CSIC and University of Valencia,
Valencia, Spain
**E-mail: acervera@ific.uv.es*

J. A. Hernando
University of Santiago de Compostela,
Santiago de Compostela, Spain

A general solution for the problem of reconstructing the evolution of a dynamic system from a set of experimental measurements is presented. This solution has been realised in a C++ toolkit that can incorporate different methods for fitting, propagation, pattern recognition and simulation. The RecPack functionality is independent of the experimental setup, what allows one to apply this toolkit to any dynamic system.

Keywords: reconstruction, kalman filter, propagation, fitting

1. Introduction

In high energy physics (HEP), as in other fields, one frequently faces the problem of reconstructing the evolution of a dynamic system from a set of experimental measurements. Most of reconstruction programs use similar methods. However, in general they are reimplemented for each specific experimental setup. Some examples are fitting algorithms (i.e. Kalman Filter[1]), equations for propagation, random noise estimation (i.e. multiple scattering), model corrections (i.e. energy loss, inhomogeneous magnetic field, etc.), etc. Similarly, the data structure (measurements, tracks, vertices, etc.), which can be generalised as well, is not reused in most of the cases.

RecPack tries to avoid that by providing a setup–independent data structure and algorithms, which can be applied to any dynamic system. The package follows an "interface" strategy, that is, all the classes that could have a different implementation have their own interface, in such a

way that the rest of the classes do not depend on such a specific implementation. This modular structure allows a great flexibility and generality.

RecPack was born in the HARP experiment at CERN-PS[2] and is currently being developed mainly for T2K.[3] Other experiments using it are MICE, MuScat, MIPP, NEMO and SuperNemo.

2. Structure of the package

RecPack distinguishes between data classes (passive) and tools (active). The tree of data classes is shown in Fig. 1. *EVector* and *EMatrix* are just a typedef of CLHEP's *HepVector* and *HepMatrix* respectively (these are vectors and matrices of *double*'s with variable dimension). This establish the only RecPack external dependence. However, the user can replace the CLHEP classes by its favorite vector and matrix classes.

A structure that appears in several levels of the data model is the pair formed by a vector of parameters (*EVector*) and its covariance matrix (*EMatrix*). Thus, a new class called *HyperVector* has being introduced to hold this repeated structure. For example, experimental measurements (*Measurement*) can be always reduced to a *HyperVector*, and the same is true for the fitting or propagation parameters (*State*). Before the track fit-

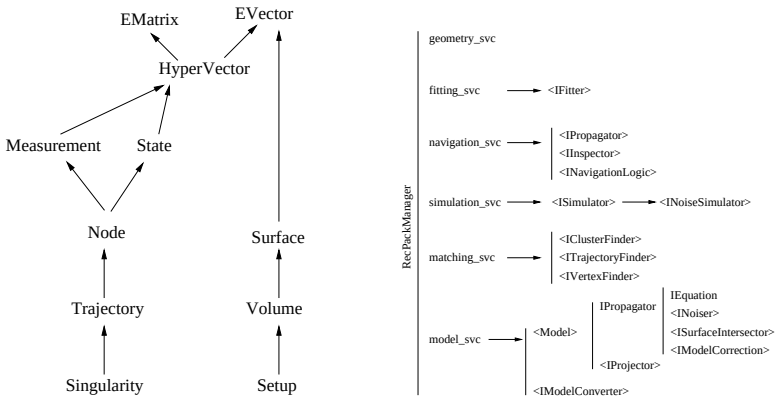

Fig. 1. Architectural design of the data classes (left) amd tools.

ting occurs, a *Trajectory* is essentially defined as a collection of uncorrelated *Measurement*'s. Track fitting results are stored in *State*'s. In the most general case, the fitting parameters are local and therefore each *Measurement* must have a *State* associated to it. An intermediate object, called *Node*,

has being introduced to accommodate the *Measurement*, the *State* and the quantities that relate both objects (the residual *HyperVector*, the local χ^2 of the fit,etc). Consequently, a *Trajectory* can be seen as a collection of *Node*'s, plus a set of global quantities as the total χ^2 of the fit, the number of degrees of freedom, etc. The classes *Surface*, *Volume* and *Setup* are treated in Sec. 3.

The tools (see Fig. 1-right) manipulate the data. Most of them are pure interfaces (hence the I), allowing them to have different implementations. Some of the tools contain sub-tools (*ISimulator, Model, IPropagator*, etc.). If a tool has several sub-tools of the same type (i.e. *Model* has several *IProjector*'s), these are stored in associative containers (<...> in the graph), which permits the access by key.

Tools are stored in services (_svc in Fig. 1), which not only actuate as containers for the tools, but also as managers. The user interacts with the *RecpackManager* class via its services. In the following sections each of the services is treated individually.

3. Geometry

The RecPack geometry service provides the methods for the definition of the experimental setup (*Setup*), which is built through the addition of volumes (*Volume*) and surfaces (*Surface*) with well defined position and axes inside the setup. One can distinguish between base surfaces, which have no boundaries, and finite surfaces, which extend the base class by incorporating a well defined size. RecPack provides some predefined volume (Box and Tube –two concentric cylinders–) and surface types (Plane: Rectangle and Ring; and Cylinder), but adding new types is straight forward. As an example, the following code defines a box called "tracker" placed inside the "mother" volume, and a surface called "wall" inside the tracker:

setup.add_volume("mother", "tracker", "box", pos, ax1, ax2, s1,s2,s3);
setup.add_surface("tracker", "wall", "rectangle", pos, ax1, ax2, s1,s2);

where pos, ax1 and ax2 are 3D *EVector*'s, and s1,s2 and s3 are doubles defining the size. In this way, complicated setups as the ones of Fig. 2 can be build.

Geometrical objects may have properties, which are indirectly associated to them in the *Setup* class. Typical volume properties are the ones that influence the evolution of the system (magnetic field, radiation length, energy loss rate, etc.). The following c++ code sets $x0$ (a *double*) as the

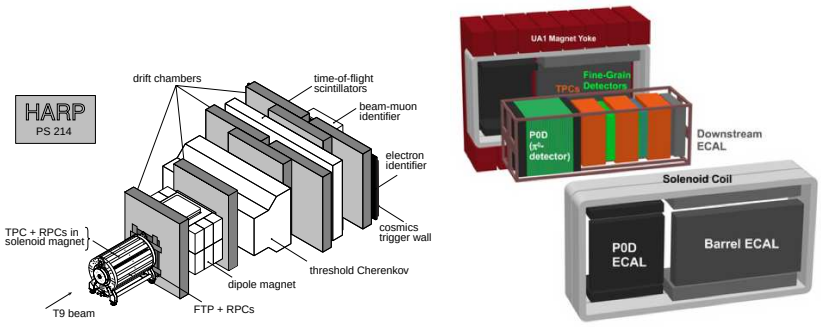

Fig. 2. Example of experimental setups corresponding to the HARP[2] detector at CERN-PS (left) and the T2K-ND280[3] detector at JPARC (right).

radiation length of the "tracker":

setup.set_volume_property("tracker", RP::X0, x0);

where $x0$ must be a data member or global variable since it is saved by reference.

4. Model

The model service is the container and manager for model related equations: intersection with surfaces, propagation and projection of states, random noise computation, etc. It contains an extensible collection of *Model*'s. Each model performs two major operations, propagation and projection:

4.1. *Propagation*

A *State* can be propagated to a given surface or length by the interface class *IPropagator*. Intermediate calculations are delegated to smaller tools:

- <*ISurfaceIntersector*>: it is a collection of *ISurfaceIntersector*'s, each of which calculates the path length to a different base surface type (Plane or Cylinder).
- *IEquation*: it computes the state vector at a given length (either provided externally or by a *ISurfaceIntersector*).
- <*IModelCorrection*>: applies a small correction to the propagation done by the *IEquation* (i.e. energy loss).
- <*INoiser*>: each of them computes the random noise covariance

matrix for the given length and for a specific type of noise (i.e. multiple scattering, energy loss fluctuations).

4.2. *Projection*

The projection operation transforms a *State* into a virtual measurement (predicted–measurement), which can be then compared with an experimental measurement to compute a residual. This is crucial for fitting and matching algorithms. The state *HyperVector* $(\vec{\mathbf{v}}, \mathbf{C}_v)$ is projected according to the following equations:

$$\vec{\mathbf{m}}^{pred} = \vec{\mathbf{h}}(\vec{\mathbf{v}}), \quad \mathbf{C}_m^{pred} = \mathbf{H}\mathbf{C}_v\mathbf{H}^T \ , \tag{1}$$

where $\vec{\mathbf{h}}$ is the projection function, which depends on the measurement type, $\mathbf{H} = \partial\vec{\mathbf{h}}/\partial\vec{\mathbf{v}}$ is the projection matrix, $\vec{\mathbf{m}}^{pred}$ is the predicted–measurement, and $\mathbf{C}_m^{pred}$ its covariance matrix.

Several measurement types ("xy", "uv", "xyz"," $r\phi$", etc.) may coexist in a single trajectory, which can be fitted to an unique model. To do so, each *Model* must contain an extensible collection of *IProjector*'s, each of which corresponds to a different measurement type.

5. Fitting

The fitting service is in charge of fitting clusters, trajectories and vertices via its fitters (*IFitter*). The user can either use one of the existing fitters or provide his own. Two fitters for trajectories ,least squares and Kalman filter,[1] are available. In the case of a Kalman Filter fit a seed state must be provided.

Fitting algorithms, called fitters, need the two setup–dependent operations described above: the prediction of the trajectory at the next measurement surface (propagaton) and the comparison betweed this prediction and the actual experimental measurement, which requires the "projection" of the predicted state to form a residual. Fitting equations can be kept independent of the model and measurement type(s) if these two operations are external to the fitter. As described above, propagation and projection are performed by each *Model*.

6. Navigation

In *Setup*'s with more than one *Volume* the propagation functionality is provided by special *IPropagator*'s, called *Navigator*'s, which are able to

handle the volume hierarchy. A default *Navigator* is provided, but others can be added easily (i.e. Geant4[4]). The default *Navigator* propagates a *State* in several steps. Propagation in each step is performed by the *IPropagator* associated to the *Model* of the *State*. Before and after each step a list of *IInspector*'s (associated to volumes and surfaces) is called. An *IInspector* is a tool that performs a concrete action: set the properties of the entering volume, sum up intermediate path lengths, set the length of the next step (dynamic stepping), etc. User defined *IInspector*'s can be added to any surface or volume.

Two important features of the default navigator are: i) the intersection with surfaces [a] is done analytically whenever is possible (and numerically otherwise) and ii) user defined *INavigationLogic*'s allow one to establish the sequence in which volumes and surfaces must be traversed.

7. Matching

This generic name refers to the methods that are related with pattern recognition (PR) problems. In general, the purpose of PR algorithms is to distribute the existing measurements into clusters, these into trajectories and these into vertices. Two types of PR algorithms are provided by RecPack: matching functions, which serve to estimate the probability of two objects of being related to each other (trajectory–trajectory, trajectory–measurement), and PR logics, which define the sequence in which such a relations are established. The first are always general, while the second may have a strong setup dependence. PR logics for *Trajectory*'s are introduced via the *ITrajectoryFinder* interface class. Currently, the RecPack matching service provides a *CellularAutomatom* trajectory finder.

8. Simulation

Some times reconstruction programs must operate over simulated measurements. However, in general the user must provide the classes and methods that allow the interface between simulation and reconstruction, which is not always an easy task. The RecPack simulation service solves this problem by generating simulated measurements with the data format required by the rest of the services.

The user must declare the active *Volume*'s and *Surface*'s (the ones that produce measurements), and specify the measurement type in each of

[a] The problem of intersecting a volume is always reduced to the intersection with its outer walls.

them. Active surfaces produce a measurement when they are intersected, while active volumes produced measurements through dynamic stepping (see Sec. 6).

Given a simulation seed (*State*), the simulation service uses the navigation service to produce an ideal trajectory inside the setup. Then, a special *IInspector* ("MeasSimulator") creates ideal measurements (according to Eq. 1) in the active volumes or surfaces by calling the *IProjector* corresponding to the measurement type in that volume or surface, and adds the propagation noise (i.e. multiple scattering) and experimental errors.

This simple simulator does not attempt to be a full simulator (i.e. Geant4). Instead, its main purpose is to serve as a debugging tool or as a fast simulator. Existing simulation toolkits, as Geant4, could be easily integrated into RecPack by implementing the *IInspector*'s that generate measurements in the different subdetectors. Such an inspectors should be able to access the Geant4 information and then use it to create specific measurements.

9. Conclusions

In summary, RecPack is a modular and extensible reconstruction toolkit, which provides the basic data structure and most of the common methods needed by any reconstruction program: matching, fitting and navigation. It also has functionality to perform a quick interface with simulation packages.

10. References

References

1. R.E. Kalman, J. Basic Eng. 82 (1960) 35
   R.E. Kalman, R.S. Bucy, J. Basic Eng. 83 (1961) 95
   R. Fruhwirth, M. Regler, Nuc. Inst. Meth. A241 (1985) 115.
   R. Fruhwirth. Nucl. Inst. Meth. A262 (1987) 444
2. http://harp.web.cern.ch/harp/
3. arXiv:1106.1238v2. Accepted for publication in Nucl. Inst. Meth. A
4. A. Dell'Acqua *et al.*, GEANT4 Collaboration, Nucl. Inst. Meth. A506 (2003) 250.