RecPack, a general reconstruction toolkit

A. Cervera-Villanueva*, J.J. Gómez-Cadenas†, J.A. Hernando‡

Abstract

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 to apply this toolkit to any dynamic system.

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]-[2]), equations for propagation, random noise estimation (i.e. multiple scattering), model corrections (i.e. energy loss, inhomogeneous magnetic field, etc.), model conversion, 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.

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 fitting occurs, a *Trajectory*

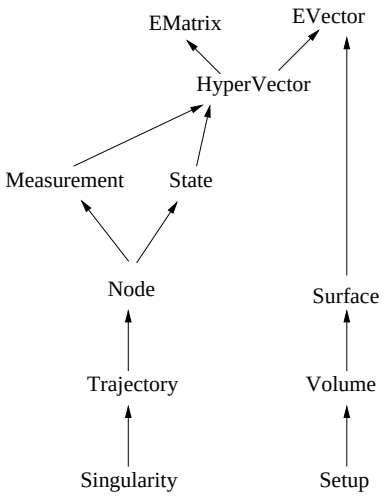

Figure 1: Architectural design of the data classes.

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 residual *Surface* [4] and the local χ^2 of the fit.). 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. Finally, two or more trajectories are connected by a *Singularity* (vertex, kink, decay), which describes their discontinuities.

The classes *Surface*, *Volume* and *Setup* are treated in Sec. GEOMETRY.

The tools (see Fig. 2) manipulate the data. Most of them are pure interfaces (hence the I), allowing them to have different implementations. Some of the tools contain subtools (*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. 2), which not only actuate as containers for the tools, but also as managers.

* Université the Genève, CH
† Universitat de Valencia, Spain
‡ CERN, Geneva, CH, and Universidade de Santiago de Compostela, Spain

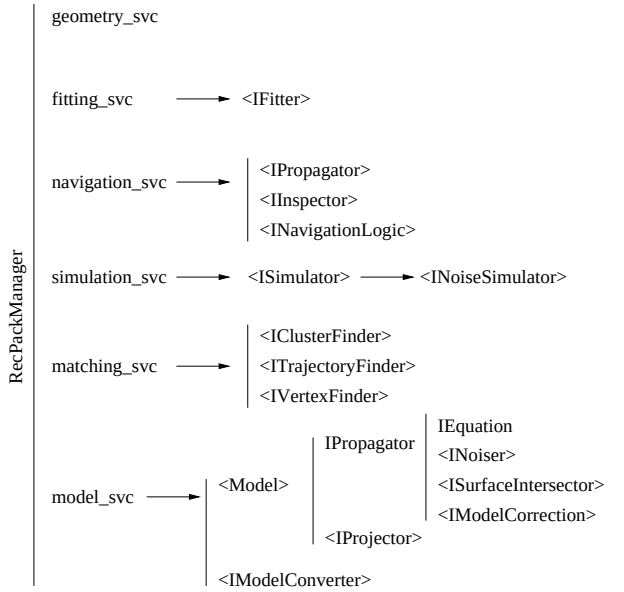

Figure 2: Architectural design of the active classes (tools). <...> means associative container.

Indeed, the user interacts with the "RecPackManager" via its services. For example, fitting a trajectory (*track*) by the least squares method to a straight line model would look like:

manager().fitting_svc().fit("Lsq", "straight_line", track);

In the following sections each of the services is treated individually.

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, tube –two concentric cylinders– and sphere) and surface types (plane: rectangle and ring; cylinder: cylinder and cylinder sector; sphere: sphere and sphere sector), 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:

```
add_volume(    "mother", "tracker", "box",
               pos, axis1, axis2, size );
add_surface(   "tracker", "wall", "rectangle",
               pos, axis1, axis2, size );
```

where pos, axis1, axis2 and size are *EVector*'s. In this way, complicated setups as the one of Fig. 3 can be build.

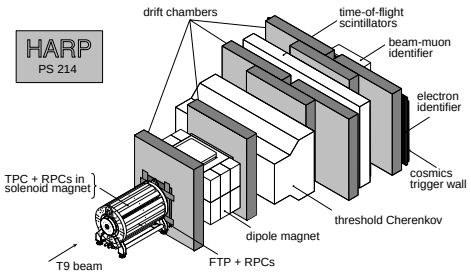

Figure 3: Example of experimental setup corresponding to the HARP detector at CERN-PS [7].

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, etc.). The following c++ code sets $x0$ (a *double*) as the radiation length of the "tracker":

set_volume_property("tracker", "X0", x0);

MODEL

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

Propagation

Propagation of states (which have a well defined position inside the setup) to a given surface or final length is performed by the interface class *IPropagator*. Intermediate calculations are delegated to smaller tools:

- *<ISurfaceIntersector>*: it is a collection of *ISurfaceIntersector*'s, each of which corresponds to a different base surface type (plane, cylinder, sphere). Its job is to calculate the path length from the actual position to the given surface.
- *IEquation*: it computes the state vector at a given length. This length can be provided by an *ISurfaceIntersector*, in the case of propagation to a surface, and externally by the user, in the case of propagation to a length.
- *<IModelCorrection>*: this kind of tool applies a small correction to the propagation than by an *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).

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. These virtual measurements may also be used by *ISimulator*'s (see Sec. SIMULATION) to produced simulated measurements. The state *HyperVector* 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, $\vec{\mathbf{m}}^{pred}$ and $\vec{\mathbf{v}}$ are the predicted–measurement vector and state vector respectively, and $\mathbf{C}_m^{pred}$ and $\mathbf{C}_v$ their corresponding covariance matrices.

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.

FITTING

Fitting algorithms, called fitters, need two setup–dependent operations: prediction of the next measurement based on the information provided by previous measurements (propagation) and comparison between real and predicted measurements (projection) to update the fitting parameters. 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*.

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]-[2]) and one for vertices (Kalman filter [3]) are available.

A trajectory fitter takes a raw *Trajectory* (a collection of *Node*'s with *Measurement*'s and empty *State*'s) and transforms it into a fitted *Trajectory*, in which the *State*'s have meaningful contents. Similarly, the vertex fitter takes a raw *Vertex* (in this case the measurements are the fitted trajectories while the state associated to each trajectory is empty before the fit). In the case of a Kalman Filter fit (for trajectories or vertices) a seed state must be provided.

The following example illustrates the functionality of this service: a set of 2D measurements have been produced by a charged particle in a magnetic field. These measurements have been already introduced in a raw *Trajectory* (*track*) and now we want to fit it, first by least squares, and then use the result of this fit (*track.state()*) as a seed for a Kalman filter fit. The necessary c++ code would be:

fit("Lsq", "Helix", track);

fit("Kalman", "Helix", track, track.state());

NAVIGATION

The main functionality of the navigation service is to propagate states to any surface, volume or length within the setup. This operation is performed by special *IPropagator*'s, called navigators, which are capable to handle the volume hierarchy and volumes with inhomogeneous properties. One navigator ("RecPackNavigator") is provided by default, but others can be added easily (i.e. Geant4 [5]).

The "RecPackNavigator" propagates the state in several steps. Propagation in each step is performed by the *IPropagator* associated to the *Model* in the actual volume. 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. For example, a "CounterInspector" could be added to a given surface in order to count the number of times this surface is traversed.

Two important features of the "RecPackNavigator" are: i) the intersection with surfaces [6] is done analytically whenever is possible (and numerically otherwise) and ii) user defined *INavigationLogic*'s allow to establish the sequence in which volumes and surfaces must be traversed.

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 trajectories and these into vertices. On can distinguish two types of PR algorithms: matching functions, which serve to estimate the probability of two objects of being related to each other (trajectory–trajectory, measurement–trajectory, trajectory–vertex, etc.), 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 are introduced via three types of tools: *IClusterFinder*, *ITrajectoryFinder* and *IVertexFinder*, which build clusters (*Measurement*'s), trajectories and vertices respectively using the available matching functions and following a specific strategy.

Currently, the RecPack matching service provides trajectory–measurement, trajectory–trajectory and trajectory–vertex matching functions. For the moment, PR logics are not implemented. In the future, one could try to identify common PR logics and include them in this service. For example, PR in a series of parallel planes which produce 2D measurements occurs always in a similar way. The same is true for a volume with 3D measurements (i.e. TPC), etc.

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 volumes and surfaces (the ones that produce measurements), and specify the measurement type in each of them. Active surfaces produce a measurement when they are intersected, while active volumes produced measurements through dynamic stepping (see Sec. NAVIGATION)

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. Finally, propagation noise and experimental errors are introduced by a set of *INoiseGenerator*'s (multiple scattering, energy loss fluctuations, measurement resolution, etc.).

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.

RECPACK VERSIONS AND CLIENTS

RecPack was born in the HARP experiment at CERN-PS [7] (see Fig. 3). The initial version, *RecPack-0* is being also used in MICE [8], MuScat [9] and MIPP [10]. A reorganization of the code, *RecPack-1* [11], was done in order to gain in flexibility and generality. This version is being used by the SciBar detector, which is part of the K2K experiment [12], and served as inspiration for LHCb [13] reconstruction software and for the design of future experiments as HERO and SuperNova [14]. The T2K [15] and NEMO [16] Collaborations have recently shown interest in using RecPack as a toolkit for their reconstruction software. A new version, RecPack-2, described in this article, is being realised at the moment for these experiments. All other RecPack users will update to the new version as soon as it is ready.

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.

ACKNOWLEDGMENTS

We would like to thank Gersende Prior for her help with the Kalman Filter vertex fit. The contribution of Malcolm Ellis and Federico Sanchez, as the main RecPack users, has being essential for reporting a non negligible amount of bugs.

REFERENCES

[1] R.E. Kalman, J. Basic Eng. 82 (1960) 35
R.E. Kalman, R.S. Bucy, J. Basic Eng. 83 (1961) 95

[2] R. Fruhwirth, M. Regler, Nuc. Inst. Meth. A241 (1985) 115.

[3] R. Fruhwirth. Nucl. Inst. Meth. A262 (1987) 444

[4] The residual is always calculated as the length of a geodesic over a surface.

[5] A. Dell'Acqua *et al.*, GEANT4 Collaboration, Nucl. Inst. Meth. A506 (2003) 250.

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

[7] http://harp.web.cern.ch/harp/

[8] http://hep04.phys.iit.edu/cooldemo/

[9] http://hepunx.rl.ac.uk/neutrino–factory/muons/muscat.html

[10] http://ppd.fnal.gov/experiments/e907/e907.htm

[11] A. Cervera-Villanueva, J.J. Gomez-Cadenas and J.A. Hernando. Nucl. Inst. Meth. A534 (2004) 180-183

[12] S.H. Ahn. et *al*. The K2K Collaboration. Phys. Lett. B511 (2001) 178-184

[13] http://lhcb.web.cern.ch/lhcb/

[14] No references available yet.

[15] http://neutrino.kek.jp/jhfnu/

[16] R. Arnold et *al*. The NEMO Collaboration. Nucl. Inst. Meth. A536 (2005) 79-122