Tracking SW Developments

Julia Hrdinka (TU Wien)
On behalf of the ACTS developers and the FCCSW team
FCC Tracking

FCC should profit from already existing software – well tested and used in running experiments

=> using the **ACTS - A(TLAS) Common Tracking Software package** for tracking

[https://gitlab.cern.ch/acts/a-common-tracking-sw](https://gitlab.cern.ch/acts/a-common-tracking-sw)

- Encapsulation of the ATLAS tracking software Code
  - updated to new coding standards
  - prepared for concurrent use (GaudiHive)

- Common tracking software toolkit useable in various applications
  - ATLAS Run3
  - FCC
  - Machine Learning Challenge 2016
ACTS – Dependencies & Build

ACTS is an experiment independent package build with Cmake

Dependencies

- Event processing framework: Gaudi
  
  
  - experiment independent
  
  - flexible structure: algorithms, services, tools
  
  - development of concurrent framework as GaudiHive
    

  => further decoupling from Gaudi under investigation

- Math library: Eigen
  

- Boost library
ACTS – package structure

[Diagram of package structure]

- Core
  - Algebra (Math library)
  - Identifier
  - CoreInterfaces (AlgorithmBase, ServiceBase, ToolBase)
  - ...

- Geometry
  - Detector
  - DetectorElementBase
  - Volumes
  - Surfaces
  - GeometryServices
  - ...

- EventData
  - Measurements
  - TrackParameters
  - ...

- Examples
  - Extrapolation
    - ExtrapolationEngine
    - RungeKuttaEngine
    - ...

- Fitting
  - KalmanFitter
  - ...

- Simulation
  - Fatras
  - Digitization
  - ...

- Plugins
  - AtlasPlugins
  - GenericDetectorPlugins
  - FCCPlugins
  - ...
ACTS – Event Data Model

Two base classes: \texttt{TrackParametrization} \& \texttt{Measurement}

- fixed size vector
- dynamic sized vector

\[(l_0, l_1, \varphi, \theta, \frac{q}{p})\]

\[
\Rightarrow \text{arbitrary (need to provide coordinate transform)}
\]

- in respect to coordinate system of specific surface (= detector element)
- using \texttt{ParameterSet} class (vector + covariance matrix) for flexible design and to customize parameters
ACTS – Extrapolation

Extrapolates the track parameters through the tracking geometry

- uses different engines to describe the sub processes
  - material effects
  - propagation
  - predictive navigation (used for Fast Track Simulation)
  - magnetic field integration (Runge Kutta Method)
- engines are stateless services for threat safety
ACTS – Fitting

Kalman Filter
- standard implementation
- reference based implementation
- possibility to recalibrate measurement

Extrapolation
- single track parameters and multi track parameters possible, supports Kalman Filter and Gaussian Sum Filter
- configurable for hole search
ACTS – Fast Simulation

Using the simplified reconstruction geometry as simulation geometry is common concept for fast simulation

The ACTS package provides

- Internal navigation through tracking geometry
- Extrapolation
  - transport through geometry & magnetic field
  - Monte Carlo material effects

=> Fast Track Simulation

- Simplified digitization
ACTS – Plugins

ACTS package is **experiment independent**

- Possibility to configure toolkit (Geometry Input, EDM, Identifier, Magnetic Field)

  => Plugins

Provided Plugins

- ATLAS Plugin
- Generic Plugin (containing detector building from python script)
- Geometry Plugins: GeoModel, TGeo (Root), DD4hep

Examples - containing geometry building tests and fast simulation

- Generic detector example
- FCC example
ACTS – Status

0.1.0 (alpha) release – done

- compiling against Gaudi (Cmake) & ATLAS environment
- first dummy detector builds with python script
- limited fast simulation available

0.2.0 (beta) release – by the end of this month

- Kalman Filter prototype
- Fast simulation for $m$, $e$, (no decay supported yet)
- Geometric digitization

Short test bed for pattern recognition - first components foreseen autumn 2016
Usage in FCC

ACTS is used as an external package

- Configure services and tools with python job option file
- Provide Identifier: DD4hep
- Provide MagneticFieldSvc (inheriting IMagneticFieldSvc)
  - currently using const magnetic field service
- DD4hep (based on Root TGeo package) used for full detector description [http://aidasoft.web.cern.ch/DD4hep](http://aidasoft.web.cern.ch/DD4hep)
  - create DD4hepDetElement inheriting from TGeoDetelement/DetectorElementBase
  - create DD4hep translation tools – particularly simplified by geometry building tools provided by ACTS

FCC geometry conversion

- Full Simulation
- Translation
- DD4hep DetectorElementBase
- Surface
- DD4hep DetectorElement
- Fast Track Simulation + Track Reconstruction
- 14/04/2016
- J.Hrdinka – TrackingSW Developemenets – FCC Week 2016
FCC(-hh) Test Tracker

Geometry of FCC-hh tracker as created with tkLayout (a toolkit for ongoing tracker design studies of CMS phase 2 upgrade tracker, adapted to FCC-hh needs) (- Z. Drasal) https://indico.cern.ch/event/438866/contributions/1085165/

Translated into DD4hep and used in ACTS

TKLayout

ACTS – Fast Simulation
FCC-hh Detector Design

with forward detectors

(-> W. Riegler)

DD4hep

ACTS – Fast simulation
Conclusion & Outlook

- First ACTS demonstrator available for FCC
  - Geometry building from DD4hep
  - Fast simulation

- Track Fitting to be integrated until the end of April
- Start of pattern recognition integration into the ACTS package in summer 2016
BACKUP SLIDES
Geometry - Basics

Tracking geometry = Simplified geometry describing sensitive material + approximated material setup

Core of the Geometry: Surface class

- represents detector element
  - connection to describing geometry via `DetectorElement`
  - base for measurement and parametrization

- layers extend surfaces
- volumes are enclosed by boundary surfaces
Geometry - Navigation

Internal Navigation

- layers contain binned array of surfaces
- volumes contain binned array of layers
- layers point to previous/next layer
- volume hierarchies staggered into each other – container volumes
- boundary surfaces of volumes point to previous/next volume
Geometry Plugins

DetectorElement class
- implement DetectorElement inheriting from the DetectorElementBase class
- serves as an interface for the Surface class to access detector specific information from the building geometry
- direct link to tracking geometry via Surface class

Translation tools
- needed to translate the geometry of the detector which is not a detector element e.g. beam tube, supporting structure
- special building tools provided by ATS simplify the translation
  - building of container volumes
  - automatic interglueing of volumes for internal navigation
  - automatic binning of layers within volumes
- Provide layer builder which builds the layers with its containing surfaces
Material Mapping of a Detector Module

- approximation of the material once per module type
- description of the material on 2D grid
- translated by automated transcript

Module A
30.4mm x 8.4mm x 0.6mm

- Copper (30.4mm x 8.4mm x 0.2mm)
- Silicon (15.2mm x 8.4mm x 0.2mm)
- Tungsten (30.4mm x 4.2mm x 0.2mm)
Example: Generic Detector

```
# the pixel module
PixelModule = DetectorModule(None, 8.4, 32.0, 0.15)

# the first layer
PixelLayer0 = CylinderLayer(PixelModule, 33., 24, 13, 0.2, 2., 0.5, 5.)
PixelLayer1 = CylinderLayer(PixelModule, 55., 40, 13, 0.2, 2., 0.5, 5.)
PixelLayer2 = CylinderLayer(PixelModule, 88., 60, 13, 0.2, 2., 0.5, 5.)
PixelLayer3 = CylinderLayer(PixelModule, 120., 72, 13, 0.2, 2., 0.5, 5.)
PixelLayer4 = CylinderLayer(PixelModule, 150., 84, 13, 0.2, 2., 0.5, 5.)

# define the pixel barrel volume
PixelBarrel = BarrelVolume( [ PixelLayer0, PixelLayer1, PixelLayer2, PixelLayer3, PixelLayer4 ] )
```
Straight line tracks through dummy detector
First Tests

- picture done with fast track simulation
- shows a mockup detector using ATS and the GenericGeometryPlugin
Dynamic vs. fixed size matrices

- ATS is using Eigen as algebra library
- What is the performance penalty when using dynamic instead of fixed size matrices?
  1) Eigen-3.2.7, g++ 4.9.2, 1M operations

<table>
<thead>
<tr>
<th></th>
<th>Dynamic sized</th>
<th>Fixed size</th>
<th>converted</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(2,5) x v(5)</td>
<td>1.7277s</td>
<td>0.797514s</td>
<td>1.45323s</td>
</tr>
<tr>
<td>M(5,2) x M(2,5)</td>
<td>3.53559s</td>
<td>2.67478s</td>
<td>3.48556s</td>
</tr>
</tbody>
</table>

2) Eigen-3.2.7, g++ 4.9.2 -O3, 100M operations

<table>
<thead>
<tr>
<th></th>
<th>Dynamic sized</th>
<th>Fixed size</th>
<th>converted</th>
</tr>
</thead>
<tbody>
<tr>
<td>M(2,5) x v(5)</td>
<td>0.154546s</td>
<td>0.00747539s</td>
<td>0.0116945s</td>
</tr>
<tr>
<td>M(5,2) x M(2,5)</td>
<td>0.217145s</td>
<td>0.031977s</td>
<td>0.0326164s</td>
</tr>
</tbody>
</table>

=> optimization compiler flags give huge speedup, fixed size operations are a factor 8-20 faster
ParameterSet class

```cpp
ParameterSet<loc1, phi> p1(0.5, 1.5*M_PI);
cout << p1->get<loc1>() << endl;  // prints 0.5
cout << p1->get<loc2>() << endl;  // gives compiler error
cout << p1->get<phi>() << endl;   // gives -0.5*M_PI
p1->getParameters();            // returns fixed size vector
p1->projector();                // returns fixed size projection matrix
```

- perform consistency checks at compile time whenever possible
  => improved performance and code stability

- provide a mechanism for checking/transforming parameter values
  caveat: at the moment only for non-local parameters

- make use of fixed size matrices/vectors whenever possible

- **fixed size** projection matrix from full parameter space created at compile time

\[
\begin{pmatrix}
  l_0 \\
  l_0 \\
  \eta \\
  \eta \\
\end{pmatrix} =
\begin{pmatrix}
  0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
  l_0 \\
  l_0 \\
  \eta \\
  \eta \\
\end{pmatrix}
\]
Measurement class

dynamic sized vector

```cpp
class MeasurementBase
{
    virtual VectorX parameters() const = 0;
};
```

```cpp
class Measurement1D: public MeasurementBase
{
    VectorX parameters() const override
    {
        return getParameters();
    }

    Vector1 getParameters() const;
};
```

```cpp
class Measurement2D: public MeasurementBase
{
    VectorX parameters() const override
    {
        return getParameters();
    }

    Vector2 getParameters() const;
};
```

fixed size vectors

- concrete measurements are of different (C++) types
  => need common base class to store concrete measurements in a single vector, right?
- measurement base class must rely on dynamically sized vectors/matrices
ATS approach: `boost::variant`

- keep performance benefit from using fixed size matrix operations while allowing to treat different concrete measurements uniformly

```cpp
enum class ParDefs { loc1, loc2, eta, phi, qop};

template<ParDefs... pars>
class Measurement
{
    typedef Vector<sizeof...(pars)> ParVector;
    ParVector parameters() const;
};

typedef boost::variant<...> FittableMeasurement;

Measurement<loc1,phi> m1;
Measurement<loc0,loc1> m2;

std::vector<FittableMeasurement> vMeasurements = {m1, m2};

struct print_visitor: public boost::static_visitor<>
{
    template<typename T>
    void operator()(const T& m) const
    {
        std::cout << "Generic measurement with " << m.parameters() << std::endl;
    }

    void operator()(const Measurement<phi>& m) const
    {
        std::cout << "Phi measurement: " << m.parameters() << std::endl;
    }
};

print_visitor v;
for(const auto& m: vMeasurements)
    boost::apply_visitor(v, m);
```
ATS binding geometry link - TrkDetElementBase

- Connects any favourite geometry module to ATS

```cpp
/** Identifier */
virtual Identifier identify() const = 0;

/** Identifier hash */
virtual IdentifierHash identifyHash() const = 0;

/** Return local to global transform */
virtual const Amg::Transform3D & transform() const = 0;

/** Return local to global transform associated with this identifier */
virtual const Amg::Transform3D & transform(const Identifier& id) const = 0;

/** Return surface associated with this detector element */
virtual const Surface & surface () const = 0;

/** Return surface associated with this identifier, which should come from the PrepRawData object (i.e. Surface s = PRD.detElement().surface( PRD.identify() ). This is only really relevant for the TRT (where there are several surfaces per detector element). For other detector elements it will just return surface() ... the Identifier is ignored. */
virtual const Surface & surface (const Identifier& id) const = 0;

/** Returns the full list of all detection surfaces associated to this detector element */
virtual const std::vector<const Surface*>& surfaces() const = 0;
```
Parameter definitions

```c
#include ATS_PARAMETER_DEFINITIONS_PLUGIN

namespace Ats
{
  enum ParDef : unsigned int
  {
    eLOC_1 = 0, /**< first coordinate in local surface frame */
    eLOC_2 = 1, /**< second coordinate in local surface frame */
    eLOC_R = eLOC_1,
    eLOC_PHI = eLOC_2,
    eLOC_RPHI = eLOC_1,
    eLOC_Z = eLOC_2,
    eLOC_X = eLOC_1,
    eLOC_Y = eLOC_2,
    eLOC_DO = eLOC_1,
    eLOC_Z0 = eLOC_2,
    ePHI = 2, /**< phi direction of momentum in global frame */
    eTHETA = 3, /**< theta direction of momentum in global frame */
    eCHARGE = 4, /**< charge/momentum for charged tracks, for neutral tracks it is 1/momentum */

    NGlobalPar /**< must be present */
  };
// end of namespace Ats
}
#endif // ATS_PARAMETER_DEFINITIONS_PLUGIN
```

- Possibility to define custom parameter definition
- Could have been: `enum class ParDef` but then `parameters[Ats::ePHI]` not possible (needs cast to `unsigned int`)
- Same for all possible parameter definitions → enforced by unit test
- Must be present
Parameter policy

```cpp
// ParameterPolicy.h

#ifndef ATS_DEFAULTPARAMETERPOLICY_H
#define ATS_DEFAULTPARAMETERPOLICY_H

// STL include(s)
#include <cmath>
#include "CoreUtils/ParameterDefinitions.h"
#include "EventUtils/ParameterTypes.h"

#ifdef ATS_PARAMETER_POLICY_PLUGIN
#include ATS_PARAMETER_POLICY_PLUGIN
#else
#endif

namespace Ats
{
    typedef ParDef ParID_t;
    typedef double ParValue_t;

template<ParID_t>
struct par_type;

template<>
struct par_type<ParDef::eLOC_1>
{
    typedef local_parameter type;
};

template<>
struct par_type<ParDef::eLOC_2>
{
    typedef local_parameter type;
};
}
```
Coordinate transformations

```c
#include ATS_COORDINATE_TRANSFORM_PLUGIN
#include ATS_COORDINATE_TRANSFORM_PLUGIN

namespace Ats {

struct coordinate_transformation {
    
    typedef AtsVector<ParValue_t, Ats::NGlobalPar> ParVector_t;

    static AtsVector<3> parameters2globalPosition(const ParVector_t& pars, const Surface& s) {
        AtsVector<3> globalPosition;
        s.localToGlobal(ParVector<3>{pars(Ats::eLOC_1), pars(Ats::eLOC_2)}, globalPosition, globalPosition);
        return globalPosition;
    }

    static AtsVector<3> parameters2globalMomentum(const ParVector_t& pars) {
        AtsVector<3> momentum;
        double p = fabs(1./pars(Ats::eQOP));
        double phi = pars(Ats::ePHI);
        double theta = pars(Ats::eTHETA);
        momentum << p * sin(theta) * cos(phi), p * sin(theta) * sin(phi), p * cos(theta);
        return momentum;
    }

```