From Simulation→Reco→Analysis

Joe Osborn

April 2, 2021

Fun4All: Now what?

- Chris gave a great overview talk on Fun4All
- Now what? How do I get analysis going?
- Where do I find information?
- What about when I have questions?

Resources for Getting Started

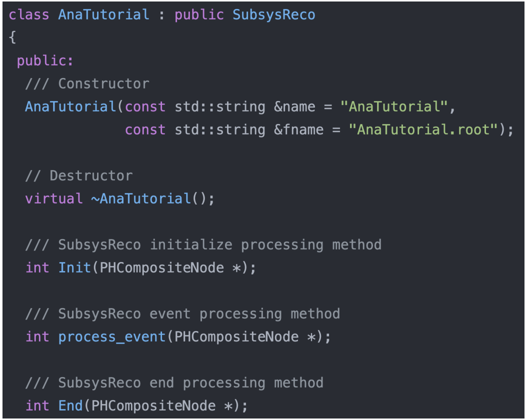
- Many resources exist to help you and guide you
  - Doxygen - code browser/documentation link
  - Github - code browser/documentation link
  - Mattermost/email communications with colleagues
    - Already available mattermost channels for ECCE on the EIC mattermost group
- Some guides and/or code that may be helpful to get you going:
  - Tutorial packages link
  - Coresoftware packages (remember, it's all Fun4All!) link
- I'll focus on the `AnaTutorial`, which is a self contained tutorial analysis package. link

Core Pieces of an Analysis

- Analysis package
    - Can be thought of as source code, or backend code
    - Does the analysis work
    - SubsysReco module(s)
    - Interacts with the node tree, etc.
- Macros
    - Runs the simulation/reconstruction/analysis
    - Tells Fun4All what to do, takes input, output, etc.

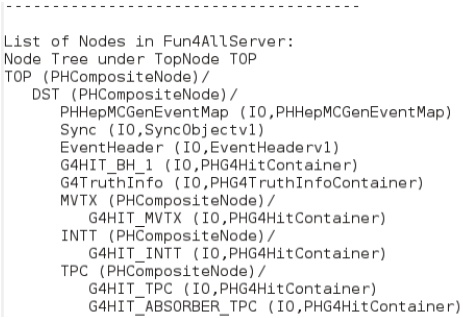
Analysis Module

- Analysis modules *must* inherit from SubsysReco base class. Tells Fun4All how to treat it
- Several methods called by Fun4All:
  - `Init(PHCompositeNode *topNode)`
  - `InitRun(PHCompositeNode *topNode)`
  - `process_event(PHCompositeNode *topNode)`
  - `ResetEvent(PHCompositeNode *topNode)`
  - `EndRun(const int runnumber)`
  - `End(PHCompositeNode *topNode)`
- Each houses the analysis code that you want to run at a given time in processing (initially, for each event, and at the end of the job)
- Take advantage of existing infrastructure, e.g. `CreateSubsysRecoModule.pl <Module Name>`



```
class AnaTutorial : public SubsysReco
{
 public:
  /// Constructor
  AnaTutorial(const std::string &name = "AnaTutorial",
              const std::string &fname = "AnaTutorial.root");

  // Destructor
  virtual ~AnaTutorial();

  /// SubsysReco initialize processing method
  int Init(PHCompositeNode *);

  /// SubsysReco event processing method
  int process_event(PHCompositeNode *);

  /// SubsysReco end processing method
  int End(PHCompositeNode *);
```

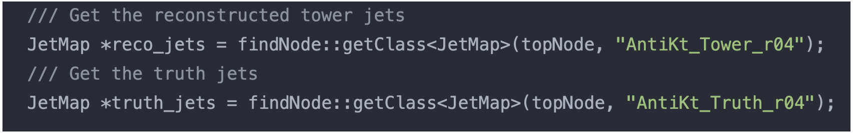
The Node Tree



```
------------------------------------

List of Nodes in Fun4AllServer:
Node Tree under TopNode TOP
TOP (PHCompositeNode)/
   DST (PHCompositeNode)/
      PHHepMCGenEventMap (IO,PHHepMCGenEventMap)
      Sync (IO,SyncObjectv1)
      EventHeader (IO,EventHeaderv1)
      G4HIT_BH_1 (IO,PHG4HitContainer)
      G4TruthInfo (IO,PHG4TruthInfoContainer)
      MVTX (PHCompositeNode)/
         G4HIT_MVTX (IO,PHG4HitContainer)
      INTT (PHCompositeNode)/
         G4HIT_INTT (IO,PHG4HitContainer)
      TPC (PHCompositeNode)/
         G4HIT_TPC (IO,PHG4HitContainer)
         G4HIT_ABSORBER_TPC (IO,PHG4HitContainer)
```

....

- The node tree is where all of the data is stored in any Fun4All job
- Users interact with the node tree to analyze, create, manipulate, data that they are interested in
- Nodes are accessed by asking the node tree



```
/// Get the reconstructed tower jets
JetMap *reco_jets = findNode::getClass<JetMap>(topNode, "AntiKt_Tower_r04");
/// Get the truth jets
JetMap *truth_jets = findNode::getClass<JetMap>(topNode, "AntiKt_Truth_r04");
```

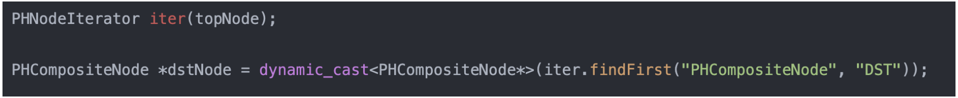
Object type (JetMap)

Node tree to search (topNode)

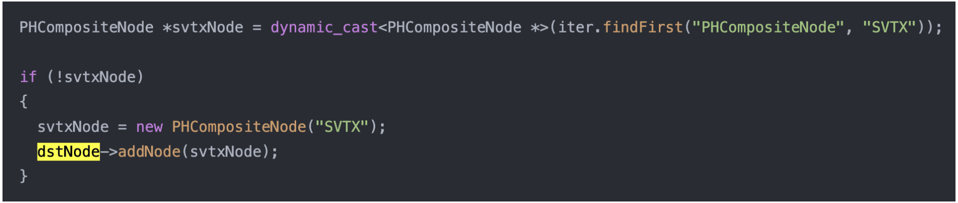
Node name on node tree (AntiKt_Truth_r04)

Nodes on Node Tree

- The beauty of Fun4All - any object can be put on the node tree
- You can create an analysis class that puts some new data structure on the node tree (e.g. a map of some arbitrary type)
- Find the subnodes you want to create a new node on:



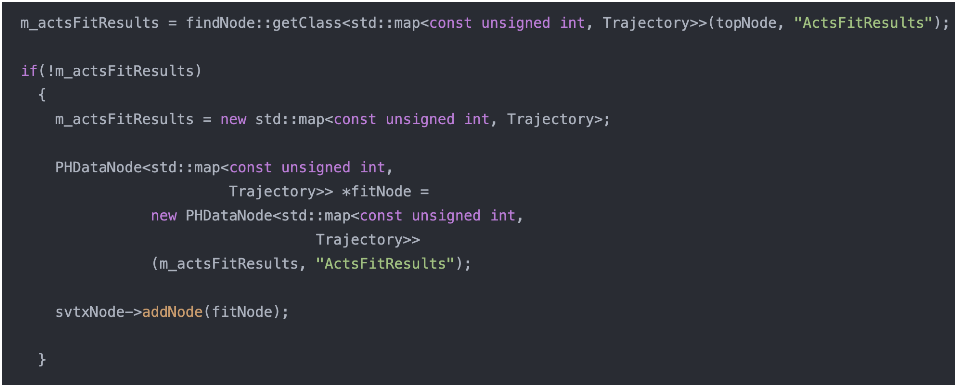
```
PHNodeIterator iter(topNode);

PHCompositeNode *dstNode = dynamic_cast<PHCompositeNode*>(iter.findFirst("PHCompositeNode", "DST"));
```



```
PHCompositeNode *svtxNode = dynamic_cast<PHCompositeNode *>(iter.findFirst("PHCompositeNode", "SVTX"));

if (!svtxNode)
{
  svtxNode = new PHCompositeNode("SVTX");
  dstNode->addNode(svtxNode);
}
```

Nodes on Node Tree

- The beauty of Fun4All - anything can be put on the node tree
- You can create an analysis class that puts some new data structure on the node tree (e.g. a map of some arbitrary type)
- Check that the object isn't already there, and if not, add it to the node tree



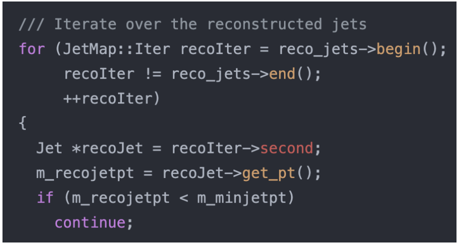
```
m_actsFitResults = findNode::getClass<std::map<const unsigned int, Trajectory>>(topNode, "ActsFitResults");

if(!m_actsFitResults)
  {
    m_actsFitResults = new std::map<const unsigned int, Trajectory>;

    PHDataNode<std::map<const unsigned int,
                        Trajectory>> *fitNode =
               new PHDataNode<std::map<const unsigned int,
                                  Trajectory>>
               (m_actsFitResults, "ActsFitResults");

    svtxNode->addNode(fitNode);

  }
```

Now what

- Now you have the tools to interact with the data nodes on the node tree
- What next?

Analysis

- With the nodes available, you can now analyze them
- Iterate over various nodes in `process_event` to get the information you want (tracks, clusters, hits, etc)
- Save analysis information in a e.g. a ROOT TTree for further analysis



```
/// Iterate over the reconstructed jets
for (JetMap::Iter recoIter = reco_jets->begin();
     recoIter != reco_jets->end();
     ++recoIter)
{
  Jet *recoJet = recoIter->second;
  m_recojetpt = recoJet->get_pt();
  if (m_recojetpt < m_minjetpt)
    continue;
```

Compiling For Fun4All

- Analysis code is compiled with a Makefile, autogen file, and configure file
- autogen is always the same, configure is always the same, Makefile has some specifics needed for your analysis package. See `AnaTutorial` for examples here
- Libraries are installed to your install directory, where all personally compiled libraries should exist (otherwise, Fun4All picks up the nightly build libraries)

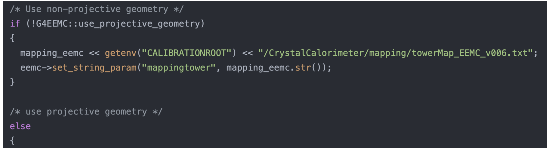
```
$ cd AnaTutorial/src
$ mkdir build
$ cd build
$ ../autogen.sh \
--prefix=/some/path/to/your/install
$ make install
```

Running Your Analysis

- You’ve written your analysis, compiled your code, and are ready to do some analysis
- Now we turn to the macros repository, which tells Fun4All what to run

Detector Macros

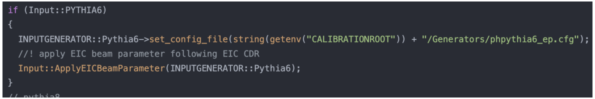
- More specific details about the detectors are stored in the `common` macros directory
- If adding a detector, or changing details about a detector, look here
- Example: EEMC details about geometry type



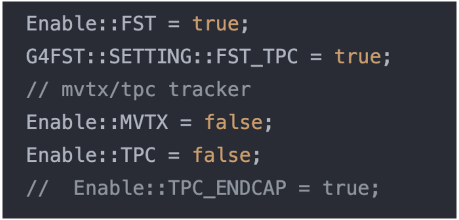
```
/* Use non-projective geometry */
if (!G4EEMC::use_projective_geometry)
{
  mapping_eemc << getenv("CALIBRATIONROOT") << "/CrystalCalorimeter/mapping/towerMap_EEMC_v006.txt";
  eemc->set_string_param("mappingtower", mapping_eemc.str());
}

/* use projective geometry */
else
{
```

Fun4All Macro

- The Fun4All macro is the conductor for your simulation job
- There are two ”sections” that you can choose to tailor to your simulation needs
  - Event generation (`Input::`)
  - Detector configuration (`Enable::`)
- Fun4All only runs what is registered with the `Fun4AllServer`, in the order it is registered
- Don’t forget to add your analysis module to Fun4All!
- Once you’re ready to run, `root.exe MyFun4AllMacro.C`



```
if (Input::PYTHIA6)
{
  INPUTGENERATOR::Pythia6->set_config_file(string(getenv("CALIBRATIONROOT")) + "/Generators/phpythia6_ep.cfg");
  //! apply EIC beam parameter following EIC CDR
  Input::ApplyEICBeamParameter(INPUTGENERATOR::Pythia6);
}
// pythia8
```



```
Enable::FST = true;
G4FST::SETTING::FST_TPC = true;
// mvtx/tpc tracker
Enable::MVTX = false;
Enable::TPC = false;
//  Enable::TPC_ENDCAP = true;
```

That’s It!

- That’s all there is to it
- Remember, Fun4All only runs what you tell it to run
- The macros are completely modular, e.g. you can create a macro that only produces simulated data, a macro that only reconstructs the simulation, etc.
- The default macro does all of this in one go, but it doesn’t have to be this way

Notes on Github Repository

- ECCE has it's own github location now: www.github.com/ECCE-EIC
- Note that there are `analysis` and `ecce-detectors` repositories
- Please use them to store and update code for analysis preservation. Has the advantages of:
  - Keeps a documented history of your code, for your own use in debugging
  - Can help others as a reference and/or they can use the code you write
  - Helps your colleagues by keeping the results you produce reproducible
- If you need some help with github workflows, there are many online resources as well as those at the following link

Last Thoughts

- There exists useful documentation online, use it
    - e.g. Recorded tutorials here, example analysis packages, etc.
- Nonetheless, don't hesitate to ask your colleagues via mattermost, email, etc.
- Happy analyzing!
- You can give the `AnaTutorial` a try right ouf of the box - take a look at the package and follow the instructions in the README