ECCE Simulations

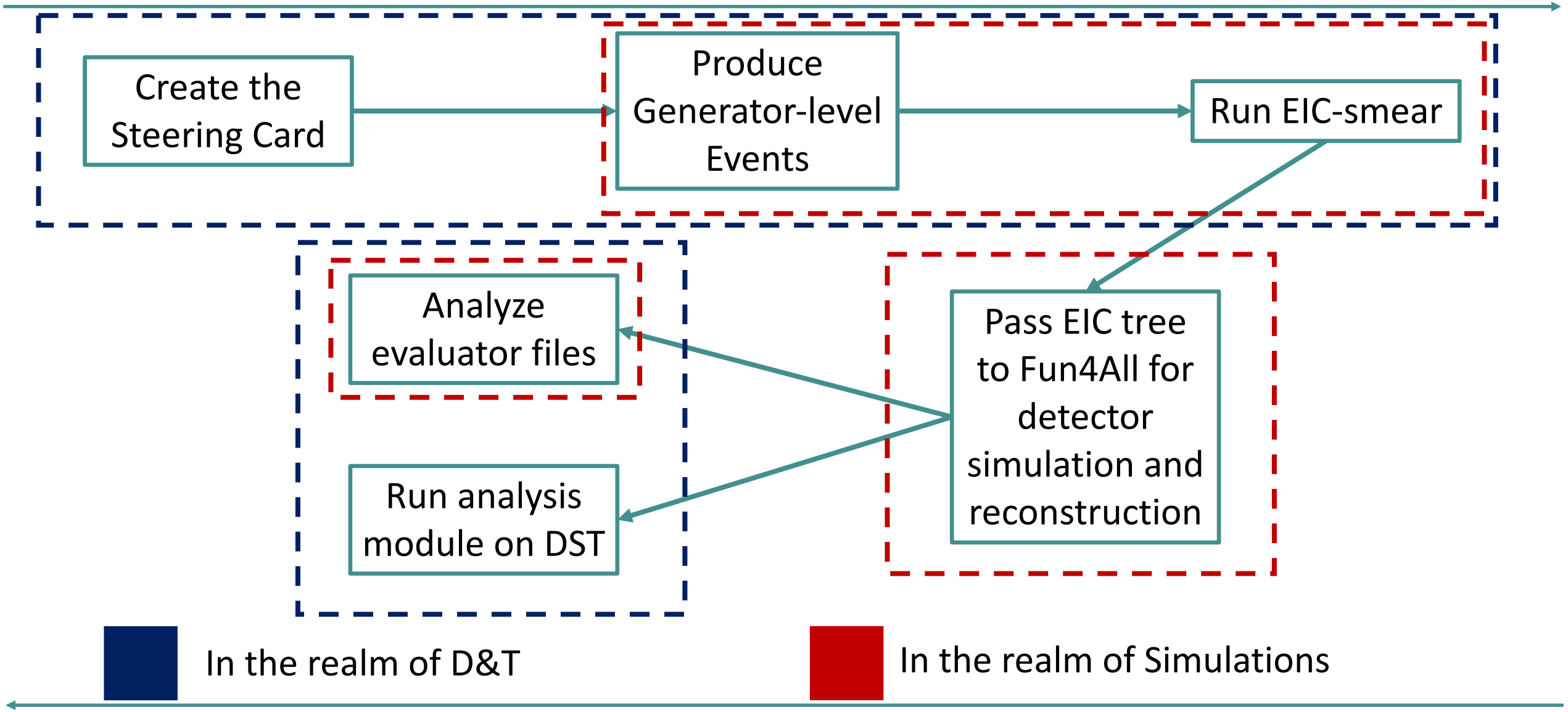
From generation to analysis

Cameron Dean, for the simulations team

05/13/2021

Diffractive and Tagging Working Group Meeting

Overview


Create the Steering Card
Produce Generator-level Events
Run EIC-smear
Analyze evaluator files
Run analysis module on DST
Pass EIC tree to Fun4All for detector simulation and reconstruction
In the realm of D&T
In the realm of Simulations

Part 0

Setting up the environment

Getting the environment

- We make nightly builds of our setup to cvmfs
- There are 3 ways to access this (4 if you have a BNL SDCC account)
  - These 3 use a singularity container

Method	Pros	Cons
Ubuntu Virtual Box	Everything is prebuilt	Shares resources with host, drivers can cause compatibility issues
Local CVMFS	Works offline	Requires large downloads and user updates
Mounted CVMFS	Is always up-to-date	Requires online connection
RCF	Direct connection to our builds on BNL farm	Needs SDCC account, graphics are difficult to configure if running on a shell

Setting up the environment

- Each method has more specific instructions but in general do:
  ```
  source /cvmfs/eic.opensciencegrid.org/ecce/gcc-8.3/opt/fun4all/core/bin/ecce_setup.sh -n
  ```
- Note: We also support tcsh
- This gets the latest build AND overwrites environment variables
  - So you won't use any local versions
- Our github repository is located here: https://github.com/ECCE-EIC

Using your own code

- If you want to modify or use your own compiled code, you will need to set the following:

```
export ECCE=/sphenix/user/cdean/ECCE
export MYINSTALL=$ECCE/install
export LD_LIBRARY_PATH=$MYINSTALL/lib:$LD_LIBRARY_PATH
export ROOT_INCLUDE_PATH=$MYINSTALL/include:$ROOT_INCLUDE_PATH
source /cvmfs/eic.opensciencegrid.org/ecce/gcc-8.3/opt/fun4all/core/bin/setup_local.sh $MYINSTALL
```

A working folder

Where to install the package

Point to the libraries and headers

Set it all up for local use

- tcsh equivalent is "`setenv MYINSTALL ${ECCE}/install`"
- Every package should have: configure.ac, Makefile.am and autogen.sh
- Build the package with:

```
mkdir build;
cd build;
../autogen.sh --prefix=$MYINSTALL;
make install;
```

Using custom macros

- If you checkout our macros repo, you can modify the files without compiling them
- You will need to set the following environment variable to see the changes though:

```
export ROOT_INCLUDE_PATH=${ECCE}/macros/common:${ROOT_INCLUDE_PATH}
```

(Assuming you downloaded the repo to $ECCE)

Part 1

Generation

Overview

- The generation is working group specific
- This is typically fast, low resource and small storage (relative)
- Can be handled directly by PWGs
- List of generators and instructions: https://eic.github.io/software/mcgen.html
- Typical generators will be pythia or BeAGLE
- This example will use BeAGLE, available at https://gitlab.in2p3.fr/BeAGLE/BeAGLE
- I will use the input file:

```
Examples/eD_18x135_Q2_1_10_y_0.01_0.95_test40k_Shd1_tau7_kt=ptfrag=0.32_shd
fac=1.32.Jpsidiffnodecay.highpf.inp
```

Running BeAGLE

- BeAGLE is built from a Makefile and then running "`make install`"
- Make an output directory with "`mkdir outForPythiaMode`"
- With BeAGLE installed run this command on one line
  (set up an environment variable pointing to the binary folder if needed)

```
./BeAGLE < Examples/eD_18x135_Q2_1_10_y_0.01_0.95_test40k_Shd1_tau7_kt=ptfrag=0.32_shdfac=1.32.Jpsidiffnodecay.highpf.inp > log.txt
```

- You will now have an output file in `outForPythiaMode` with all your generated events:
  `outForPythiaMode/collision_Q2_min_max_y_min_max_tau_val_noquench_kt=ptfrag=0.32_Shd1_ShdFac=1.32_diffractive_Jnodecay_testnumev.txt`

Running EIC-smear

- We can turn the text file into a smeared TTree, readable by Fun4All with the following commands:

```
cd outForPythiaMode/
root -l
gSystem->Load("libeicsmear.so");
BuildTree("collision_Q2_min_max_y_min_max_tau_val_noquench_kt=ptfrag=0.32_Shd1_ShdFac=1.32_diffractive_Jnodecay_testnumev.txt", ".", -1, "log.txt")
```

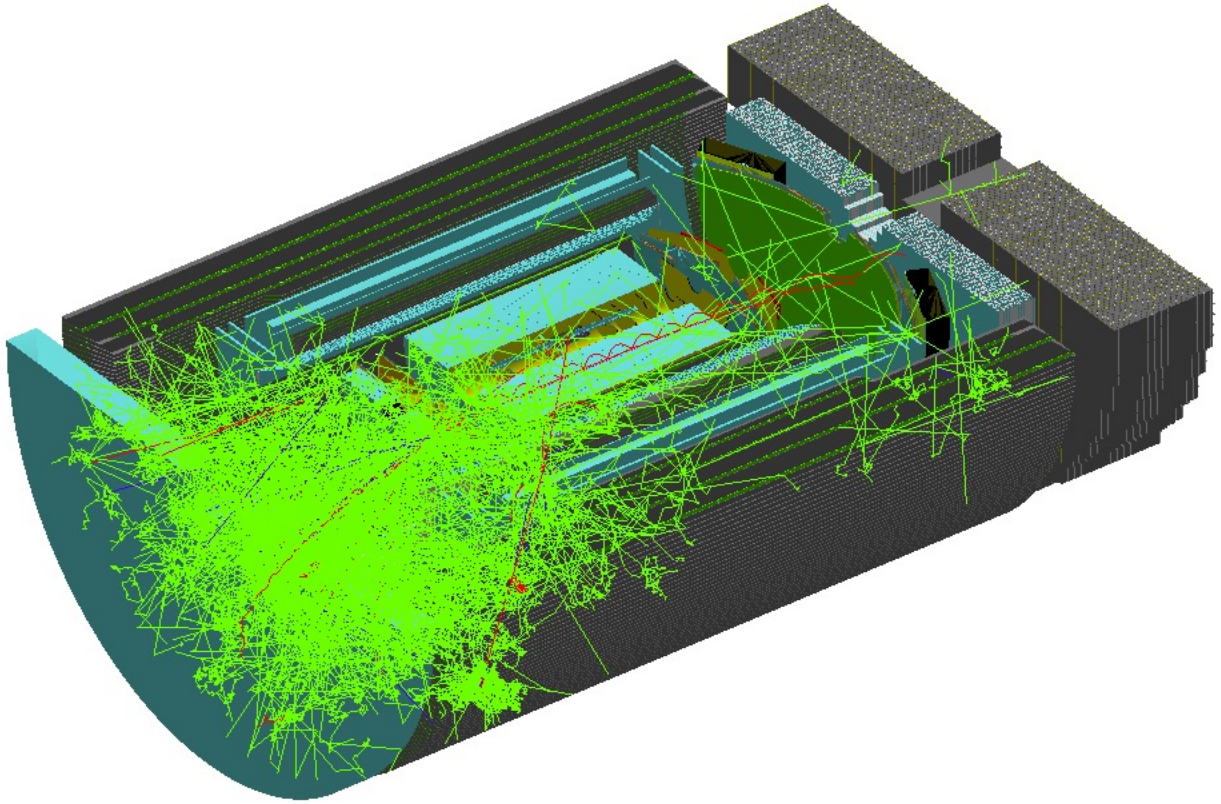
- The BuildTree options are:

```
BuildTree(“<inputFile>”, “<outputPath>”, <nEvents>, “<logFile>”)
```

- You should now have an output tree that we can run the simulation on

Part 2

Simulation



Overview

- Our detector simulation is located in `macros/detectors/EICDetector/Fun4All_G4_EICDetector.C`
- This macro can read in an EIC TTree, simulate detector responses, do the event reconstruction, write the event information to files (DST or TTree) and visualize the detector/event
- This is a top-level macro that runs smaller macros in `macros/common` and `G4Setup_EICDetector.C`
- Default action is pion injection by setting `Input::Simple = true`
- Display is **disabled** by default
- DST writing is **disabled** by default
- Evaluators are **enabled** by default

Building your own detector

- In this example, we will build and run an event through a VERY simple detector
- This will be done locally in the detector folder but if you set up the macros environment variable you can pick this macro up from the common folder
- We will build a central barrel detector, 2m long, 20cm thick and 10cm from the beam axis. This will be a silicon detector
- Start by creating an empty file, **G4_SimpleDetector.C**, in macros/detectors/EICDetector

Includes

- The headers and libraries you need are:

```
#ifndef MACRO_G4SIMPLEDETECTOR_C
#define MACRO_G4SIMPLEDETECTOR_C
#include <GlobalVariables.C>
#include <g4detectors/PHG4CylinderSubsystem.h>
#include <g4main/PHG4Reco.h>
R__LOAD_LIBRARY(libg4detectors.so)
```

Namespaces

- We will add some namespaces to enable the detector or change properties:

```
namespace Enable
{
  bool SIMPLEDETECTOR = false;
  bool SIMPLEDETECTOR_ABSORBER = false;
  bool SIMPLEDETECTOR_OVERLAPCHECK = false;
  int SIMPLEDETECTOR_VERBOSITY = 0;
}  // namespace Enable

 namespace G4SIMPLEDETECTOR
{
  double simpledetector_radius = 10.0;
  double simpledetector_thickness = 20.0;
  double simpledetector_length = 200.0;
}  // namespace G4SIMPLEDETECTOR
```

Positioning the detector

- We can initialize the detector with some basic positioning

```
void SimpleDetectorInit()
{
  BlackHoleGeometry::max_radius = std::max(BlackHoleGeometry::max_radius,
G4SIMPLEDETECTOR::simpledetector_radius);
  BlackHoleGeometry::max_z = std::max(BlackHoleGeometry::max_z,
G4SIMPLEDETECTOR::simpledetector_length / 2. + no_overlapp );
  BlackHoleGeometry::min_z = std::min(BlackHoleGeometry::min_z, -
(G4SIMPLEDETECTOR::simpledetector_length / 2.) - no_overlapp);
}
```

Calculate the particle response

- We now create a function that returns a double, this calculates the position of the particle as it transitions the detector based on the previous position:

```
double SimpleDetector(PHG4Reco* g4Reco, double radius)
{
  bool AbsorberActive = Enable::ABSORBER || Enable::SIMPLEDETECTOR_ABSORBER;
  bool OverlapCheck = Enable::OVERLAPCHECK || Enable::SIMPLEDETECTOR_OVERLAPCHECK;
  int verbosity = std::max(Enable::VERBOSITY, Enable::SIMPLEDETECTOR_VERBOSITY);

  if (radius > G4SIMPLEDETECTOR::simpledetector_radius)
  {
    cout << "inconsistency: radius: " << radius
         << " larger than simpledetector inner radius: " << G4SIMPLEDETECTOR::simpledetector_radius
         << endl;
    gSystem->Exit(-1);
  }
```

//We will continue to add code on the next several slides

Calculate the particle response

- This is an empty space cylinder up to the inner radius of the detector:

```
//Create a vacuum until the detector
PHG4CylinderSubsystem* cyl = new PHG4CylinderSubsystem("VAC_SIMPLEDETECTOR", 0);
cyl->set_double_param("radius", 0.0);
cyl->set_int_param("lengthviarapidity", 0);
cyl->set_double_param("length", G4SIMPLEDETECTOR::simpledetector_length);
cyl->set_string_param("material", "G4_Galactic");
cyl->set_double_param("thickness", G4SIMPLEDETECTOR::simpledetector_radius);
cyl->SuperDetector("SIMPLEDETECTOR");
if (AbsorberActive) cyl->SetActive();
cyl->OverlapCheck(OverlapCheck);
g4Reco->registerSubsystem(cyl);
```

Calculate the particle response

- Now we create our active volume:

```
cyl = new PHG4CylinderSubsystem(”SI_SIMPLEDETECTOR", 1);
cyl->set_double_param("radius", G4SIMPLEDETECTOR::simpledetector_radius);
cyl->set_int_param("lengthviarapidity", 0);
cyl->set_double_param("length", G4SIMPLEDETECTOR::simpledetector_length);
cyl->set_string_param("material", "G4_Si");
cyl->set_double_param("thickness", G4SIMPLEDETECTOR::simpledetector_thickness);
cyl->SuperDetector("SIMPLEDETECTOR");
if (AbsorberActive) cyl->SetActive();
cyl->OverlapCheck(OverlapCheck);
g4Reco->registerSubsystem(cyl);
```

Calculate the particle response

- Now we finish the radius calculation and close the function:

```
   radius = G4SIMPLEDETECTOR::simpledetector_radius + G4SIMPLEDETECTOR::simpledetector_thickness;
   radius += no_overlapp;

   return radius;
}
#endif
```

- Save and exit

Adding your detector to the simulation

- Inside **G4Setup_EICDetector.C** do the following:
- Add the header:
  `#include “G4_SimpleDetector.C”`
  (note “” is for local files and <> is for external files)
- Inside `void G4Init()` add:

```
if (Enable::SIMPLEDETECTOR) SimpleDetectorInit();
```

- Inside `int G4Setup()` add:

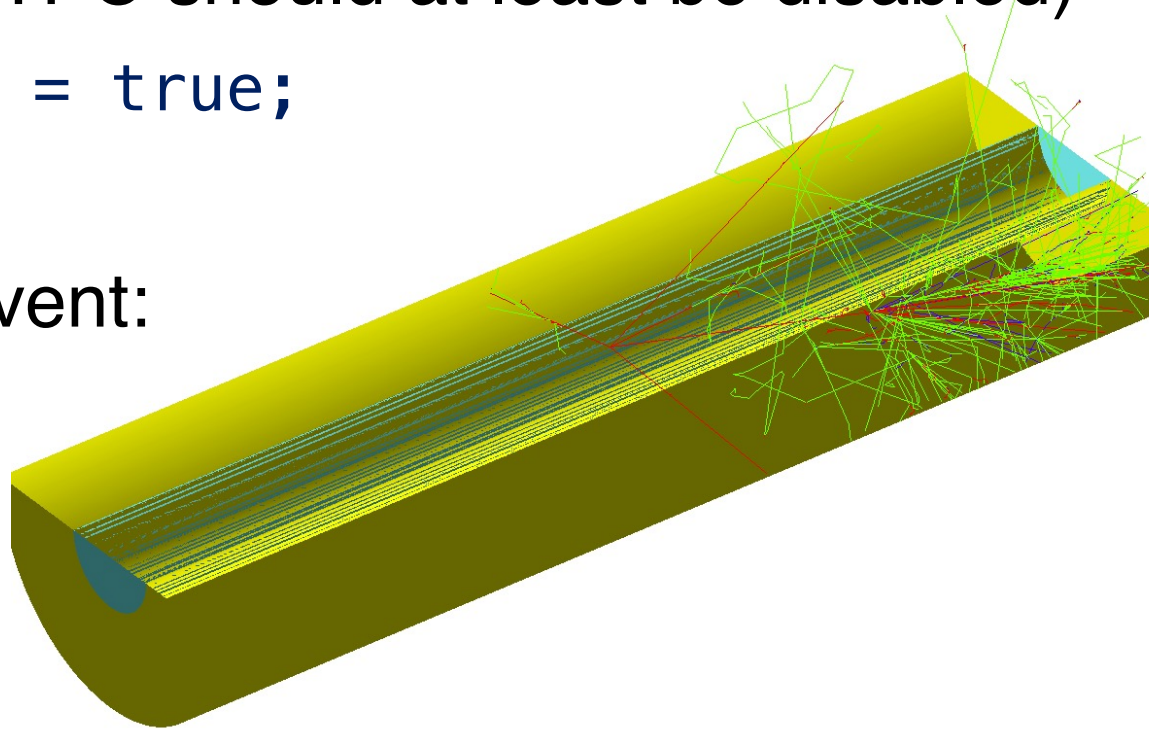
```
if (Enable::SIMPLEDETECTOR) radius = SimpleDetector(g4Reco, radius);
```

- Save and close G4Setup_EICDetector.C

Adding your detector to the simulation

- Open **Fun4All_G4_EICDetector.C**
- Set `Enable::DISPLAY = true;`
- Disable any other detectors you wish (TPC should at least be disabled)
- Add the line `Enable::SIMPLEDETECTOR = true;`
- Save and exit
- Now run the macro and generate an event:

```
root –l Fun4All_G4_EICDetector.C
se->run(1)
```



Part 3

Analysis

Studying the output

- There are two ways to study the output of the files:
  1. Using the evaluators
  2. Writing an analysis module
- The first way will be covered in a dedicated talk at the next software workshop (21st of May)
  - These files can be written out during the production if needed and directly analyzed
- The second option requires writing classes that can run in Fun4All
  - We have a user macro that can be enabled in Fun4All_EICDetector.C, just add your module here
  - It is enabled in Fun4All_EICDetector.C here

Creating a module

- Each module has 3 basic parts: initialization, processing and end.
- Say we want to make a module named **AnalysisModule.cc** (we also would have an associated header). The basic file would be:

```
#include “AnalysisModule.h”
#include <fun4all/ Fun4AllReturnCodes.h>
#include <phool/getClass.h>

AnalysisModule::AnalysisModule(){} #Constructor

AnalysisModule::Init(PHCompositeNode *topNode)
{return 0;}

AnalysisModule::process_event(PHCompositeNode *topNode)
{return Fun4AllReturnCodes::EVENT_OK;}

AnalysisModule::End(PHCompositeNode *topNode)
{return 0;}
```

Available return codes are defined here
The codes can be used to control event processing behavior such as skipping the event

There also exists ResetEvent, `InitRun` and EndRun

Doing something in the module

- Let's say we want to figure out the x-momentum of each track in the event
- Inside `AnalysisModule::process_event` we could have:

```
SvtxTrackMap *m_dst_trackmap = findNode::getClass<SvtxTrackMap>(topNode, "TrackMap");


for (SvtxTrackMap::Iter iter = m_dst_trackmap->begin(); iter != m_dst_trackmap->end();
++iter)
{
  SvtxTrack *m_dst_track = iter->second;
  std::cout << "The tracks x-momentum is " << m_dst_track->get_px() << " GeV" << std::endl;
}
```

- Our doxygen is a good place to look at our classes and member functions
  - SvtxTrack is detailed here

AnalysisModule.h

- The structure of the header would look something like:

```
#ifndef ANALYSISMODULE_H
#define ANALYSISMODULE_H

#include <trackbase_historic/SvtxTrackMap.h>
#include <trackbase_historic/SvtxTrack.h>
#include <fun4all/SubsysReco.h>

class AnalysisModule : public SubsysReco

Public:
  AnalysisModule();
  ~AnalysisModule();
  int Init(PHCompositeNode *topNode);
  int process_event(PHCompositeNode *topNode);
  int End(PHCompositeNode *topNode);

#endif //ANALYSISMODULE
```

Building the package

- You can generate a template module using:

```
CreateSubsysRecoModule.pl <package name>
```

- To build the package you need: `configure.ac, Makefile.am and autogen.sh`
- Look at coresoftware for examples on how to set these up
  - autogen.sh does not need to be changed
  - configure.ac will have one line to define the subfolder your includes go in
  - Makefile.am is specific to your package
- **Makefile.am** could have:

```
lib_LTLIBRARIES = libanalysismodule.la
pkginclude_HEADERS = AnalysisModule.h
libanalysismodule_la_SOURCES = AnalysisModule.cc
```

Running the package

- After you've compiled the package following the instructions in Part 0 you can run your package in **`G4_User.C`** from the following:
- Add your header and library to the macro:

```
#include <AnalysisModule.h>
R__LOAD_LIBRARY(libanalysismodule.so)
```

- Now register your module with Fun4All:

```
void UserAnalysisInit()
{
  Fun4AllServer* se = Fun4AllServer::instance();
  AnalysisModule *myAnalysisModule = new AnalysisModule();
  se->registerSubsytem(myAnalysisModule);
  return;
}
```

Conclusions

- Please use these slides as a reference for your analysis development, they cover everything from generation, simulation, detector building right up to physics analysis
- We have lots of references to help your development:
  - Doxygen: https://ecce-eic.github.io/doxygen/
  - Github: https://github.com/ECCE-EIC/
  - Software Documentation: https://ecce-eic.github.io/
  - Simulation Office Hours: https://indico.bnl.gov/event/11574/
- There is an upcoming simulation workshop where we will cover this all in more detail over a longer period than this talk.
  - Please register here