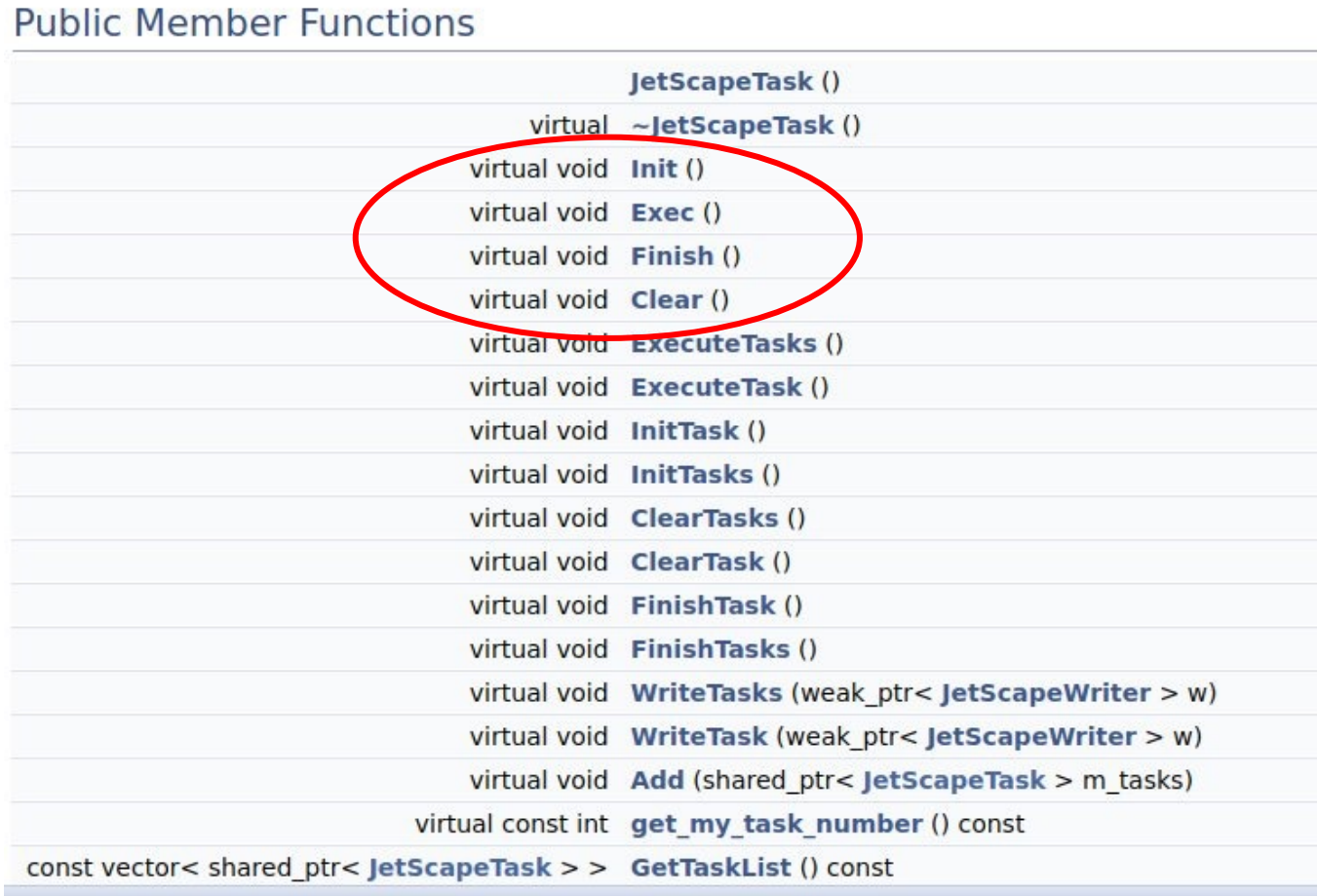
Reminder:Task-Based Framework And Adding a *Hello World Task/Module*

Task-Based Framework

- JETSCAPE Framework provides base classes
  - JetScapeTask
    - All the tasks in the framework are subclasses of this class
    - Methods called recursively by framework
      - Init()
      - Exec()
      - Finish()
      - Clear()
    - Each task must implement these methods



Public Member Functions

	JetScapeTask ()
virtual	**~JetScapeTask** ()
virtual void	**Init** ()
virtual void	**Exec** ()
virtual void	**Finish** ()
virtual void	**Clear** ()
virtual void	**ExecuteTasks** ()
virtual void	**ExecuteTask** ()
virtual void	**InitTask** ()
virtual void	**InitTasks** ()
virtual void	**ClearTasks** ()
virtual void	**ClearTask** ()
virtual void	**FinishTask** ()
virtual void	**FinishTasks** ()
virtual void	**WriteTasks** (weak_ptr< **JetScapeWriter** > w)
virtual void	**WriteTask** (weak_ptr< **JetScapeWriter** > w)
virtual void	**Add** (shared_ptr< **JetScapeTask** > m_tasks)
virtual const int	**get_my_task_number** () const
const vector< shared_ptr< **JetScapeTask** > >	**GetTaskList** () const

Add Your Module: Hello World!

- Add the header file
- overrides JetScapeTask
  - Methods are virtual
  - Will be implemented in source file

To add you own task or module code:

- **Create a header file *.h**
- **Create a source file *.cc**

You can add these test files in the framework folder. If your code ends with .cc then cmake will automatically find and compile!

```
#ifndef HELLOWORLDMODULETEST_H
#define HELLOWORLDMODULETEST_H
#include "JetEnergyLossModule.h"
using namespace Jetscape;
class HelloWorld : public JetScapeTask
{
public:
HelloWorld();
virtual ~HelloWorld();
virtual void
virtual void Init();
virtual void Exec();
virtual void Clear();
virtual void Finish();
};
#endif
```

Add Your Module: Hello World!

- Add the source file
- Implements JetScapeTask methods

```
#include "HelloWorldModuleTest.h"
#include<iostream>

using namespace Jetscape;

HelloWorld::HelloWorld()
{
   SetId("HellowWorld");
   VERBOSE(8);
}

HelloWorld::~HelloWorld()
{
   VERBOSE(8);
}

void HelloWorld::Init()
{
   INFO<<"Initialize HelloWorld Module ...";
}

void HelloWorld::Exec()
{
    INFO<<"This is the Module Executing... HELLO WORLD!...";
}

void HelloWorld::Clear()
{
    INFO<<"This is the Module Clearing...";
}

void HelloWorld::Finish()
{
    INFO<<"This is the Module Finishing...";
}
```

Add Your Hello World Module in Brick test

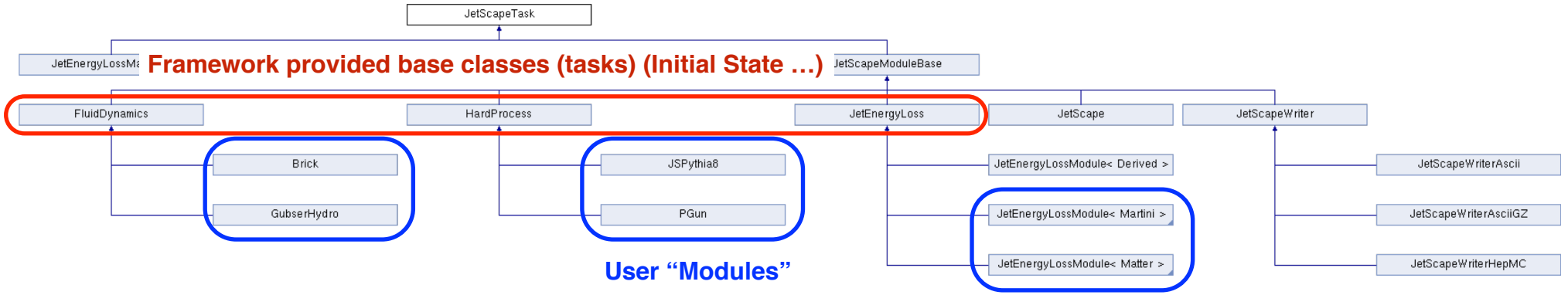
- Include the header file
  - `#include "HellowWorldModuleTest.h"`
- Create a pointer
  - `auto helloWorld = make_shared<HelloWorld> ();`
- Attach the pointer to the main task
  - `jetscape->Add(helloWorld);`
- Framework will run your module

```
[Verbose][7] 0MB virtual void Jetscape::JetScapeTask::ExecuteTasks() :  : # Subtasks = 0
[Info] 0MB  This is the Module Executing... HELLO WORLD!...
[Info] 0MB  Run JetScapeWriterAscii: Write event # 9 ...
[Verbose][7] 0MB virtual void Jetscape::JetScapeTask::ClearTasks() :  : # Subtasks = 5
[Info] 0MB  This is the Module Clearing...
[Info] 0MB  JetScape finished after 10 events!
```

Adding you own physics module/task

Similar to the hello world task example since all physics modules are also JetScapeTasks …

Generically: How to create your own physics module?



Derive your physics module from the JetScape framework provided base classes:
FluidDynamics, HardProcess, JetEnergyLoss, InitialState …

These base classes provide the ***interface and data types*** (defined by the JetScape framework)
—> **Override** these interface functions following the task based approached:
Init(), Exec(), Finish() … (some modules require to override different functions according to their physics needs. A comprehensive manual will be provided soon…)

<u>Two different approaches concerning your physics code:</u>
a) Implement your **code directly** in your derived JetScape class
b) Use the derived JetScape class as a **wrapper** to your code (either source or library)

In more detail: Hydro and Jet Energy Loss Interface …

```
class MyBrick: public FluidDynamics {

 private:
  …

 public:

    MyBrick();
     ~MyBrick();
    …

    void evolve_hydro();
    void get_hydro_info(real t, real x, real y, real z,
                        FluidCellInfo* fluid_cell_info_ptr);
    …

    void InitTask();

    …
};
```

Derive from hydro base class

Override to execute your hydro code and allow access to hydro info …

Override to initialize your hydro code …

Following Task approach:
Init() —> InitTask()
Exec() —> evolve_hydro()
Clear() and Finish() if needed can be implemented!

Derive from JetEnergyLoss base class with < … > template to allow cloning

```
class Matter : public JetEnergyLossModule<Matter> {

 public:

  Matter();
  virtual ~Matter();

  void DoEnergyLoss(double deltaT,double time, double Q2,
                    vector<Parton>& pIn, vector<Parton>& pOut);

  void WriteTask(weak_ptr<JetScapeWriter> w);
  …

  void Init();
  …

 protected:
  uniform_real_distribution<double> ZeroOneDistribution;

};
```

Override the jet energy loss slot (from signal/slot) to handle n->m parton energy loss

Override to initialize your jet energy loss code …

Following “Task” approach:
Init() —> Init()
“Exec()” —> DoEnergyLoss(…) (on parton not event level)
Clear() and Finish() if needed can be implemented!

Reminder: Init/InitTask and XML … (JetScapeXML instance)

```
class Matter : public JetEnergyLossModule<Matter>
{
 public:

  Matter();
  virtual ~Matter();
  void Init();

  …
};
```

```
// As example: Toy Matter Eloss
void Matter::Init()
{
  INFO<<"Intialize Matter ...";

  tinyxml2::XMLElement *eloss= JetScapeXML::Instance()->GetXMLRoot()->FirstChildElement("Eloss" );
  tinyxml2::XMLElement *matter=eloss->FirstChildElement("Matter");

  if (matter)
    {
      string s = matter->FirstChildElement( "name" )->GetText();

      DEBUG << s << " to be initilizied ...";

      double m_qhat=-99.99;
      matter->FirstChildElement("qhat")->QueryDoubleText(&m_qhat);
      SetQhat(m_qhat);

      DEBUG  << s << " with qhat = "<<GetQhat();
    }
  else
    {
      WARN << " : Matter not properly initialized in XML file ...";
      exit(-1);
    }
}
```

```
<?xml version="1.0"?>
<!-- Just for test purposes! -->
<!-- More details and final format to be determined ... -->
<jetscape>

  <debug> on </debug>
  <remark> off </remark>
  <vlevel> 0 </vlevel>

  <!--  If for example one wants to run JetScape purely via xml ... -->
   <Tasks>
     <task>Matter</task>
     <task>Hydro1</task>
   </Tasks>

   <!-- Inital State Module ... -->
   <IS>

     <Trento>
     </Trento>

                                                                    IS pos = 0 ... -->

                                                                    loss Manager ... -->
                                                                     in Matter and Martini ... -->

     </Martini>

   </Eloss>
```

Adding your code/new class

To add you own physics module code:

- **Create a header file *.h**
- **Create a source file *.cc**

You can add these test files
in the jet/hydro/initialstate folder.
If your code ends with .cc then cmake
will automatically find and compile!

Execute/Test:

- Include the header file
  - `#include “MyModule.h"`
- Create a pointer
  - `auto myModule = make_shared<MyModule> ();`
- Attach to main task or JetEnergyLoss via JetScapeTask Add()
  - `xxx->Add(myModule);`
- Framework will run including your module

Using JETSCAPE Output

Option 0: Pythia8 Style

- Convince developers (us) that the following is an important functionality:

```
// Run JetScape with all task/modules as specified ...
// jetscape->Exec();
while ( jetscape->Next() ){
  JetScapeEvent event=jetscape->Event();
  vector<JetScapeParticleBase> particles = event->GetFinalParticles();
  //
}
```

NOT valid code!!

- No such thing (yet) as a JetScapeEvent class
- Should have the final state particles, but what about hydro, initial state, full shower, …?

- **Breaks task paradigm**, while being functionally all but identical to the more compliant Options 1&2

Option 1: Create an Analyzer class

- Derive from **`JetScapeWriter`**
- Override relevant methods **`Write(), WriteEvent`**
- Attach to jetscape object

- Powerful, elegant
- Direct access to shower and (eventually) hydro, IS, … info

- Untested. **`JetScapeWriter`** class has a few quirks

Please try and report back! ☺

```
#include "JetScapeWriter.h"

namespace Jetscape {
class Analyzer : public JetScapeWriter
{

 public:

  Analyzer() {};
  virtual ~Analyzer();

  void Init();
  void Exec();

  void Write(weak_ptr<Parton> p);
  void Write(weak_ptr<Vertex> v);
  void WriteEvent();

  // ...
 private:
  std::ofstream output_file; //!< Output file
  //...

};
} // end namespace Jetscape
```

```
auto analyzer= make_shared<Analyzer> (outname);
jetscape->Add(analyzer);
```

Option 2: Experimentalist Workflow

1. Create a large chunk of data
2. Analyze data in a separate step

- Current output formats
  - HepMC
  - **ASCII**
  - **Gzipped ASCII**

- Plans for Rivet
- Your input is welcome!

src/test/readerTest.cc

```
vector<shared_ptr<PartonShower>> mShowers;

// open output in ascii format
auto reader=make_shared<JetScapeReaderAscii>("test_out.dat");

// reads in multiple events and multiple showers per event
while (!reader->Finished())
 {
   reader->Next();

   cout<<"Analyze current event = "<<reader->GetCurrentEvent()<<endl;
   mShowers=reader->GetPartonShowers();

   for (int i=0;i<mShowers.size();i++)
    {
      cout<<" Analyze parton shower = "<<i<<endl;

      // Access to initial parton
      cout<<"Shower initiating parton : "<<*(mShowers[i]->GetPartonAt(0))<<endl;

      // Access to final partons
      vector<shared_ptr<Parton>> partons = GetFinalPartons();

      // FastJet style access
      fjcore::JetDefinition jet_def(fjcore::antikt_algorithm, 0.4);
      fjcore::ClusterSequence cs(mShowers[i]->GetFinalPartonsForFastJet(), jet_def);
      vector<fjcore::PseudoJet> jets = fjcore::sorted_by_pt(cs.inclusive_jets(2));
 }
reader->Close();
```

A Crash Course in ROOT

De-facto standard in HEP-EX for data analysis
https://root.cern.ch/guides/users-guide

`src/test/readerTestWithRoot.cc`

```
#include <TH1.h>
#include <TFile.h>

// ...

int main(int argc, char** argv) {

  // Open output root file
  TFile* fout = new TFile( "test_out.root", "RECREATE");

  // turn on error bars by default
  TH1::SetDefaultSumw2();

  // Create histograms
  TH1D* jetpt = new TH1D ("jetpt","Jet p_{T};p_{T}^{jet} [GeV/c];counts",120, 0, 120);

   // ...
   while (!reader->Finished())    {
     reader->Next();

         // Actual jetfinding
         fjcore::ClusterSequence cs(mShowers[i]->GetFinalPartonsForFastJet(), jet_def);
         vector<fjcore::PseudoJet> jets = fjcore::sorted_by_pt(cs.inclusive_jets(2));

         // Fill histograms
         for (int k=0;k<jets.size();k++){
           // cout<<"Anti-kT jet "<<k<<" : "<<jets[k]<<endl;
           jetpt->Fill( jets[k].pt() );
         }

       }
    }
  reader->Close();
  fout->Write();
}
```

Include relevant headers

ROOT file

THx: x-dimensional histogram

TH1D: double precision

Constructor accepts:

- Name (ensure uniqueness…)
- Title and axis titles
- Range and number of bins

Fill histogram, i.e. increase bin content by 1 (or a given weight)

Save all ROOT objects for later visualization

Let's do some "Physics"!

Using the standard Matter module,

1. Create O(100) events with qhat=0 → vacuum
2. Create O(100) events with qhat=5 → extreme medium

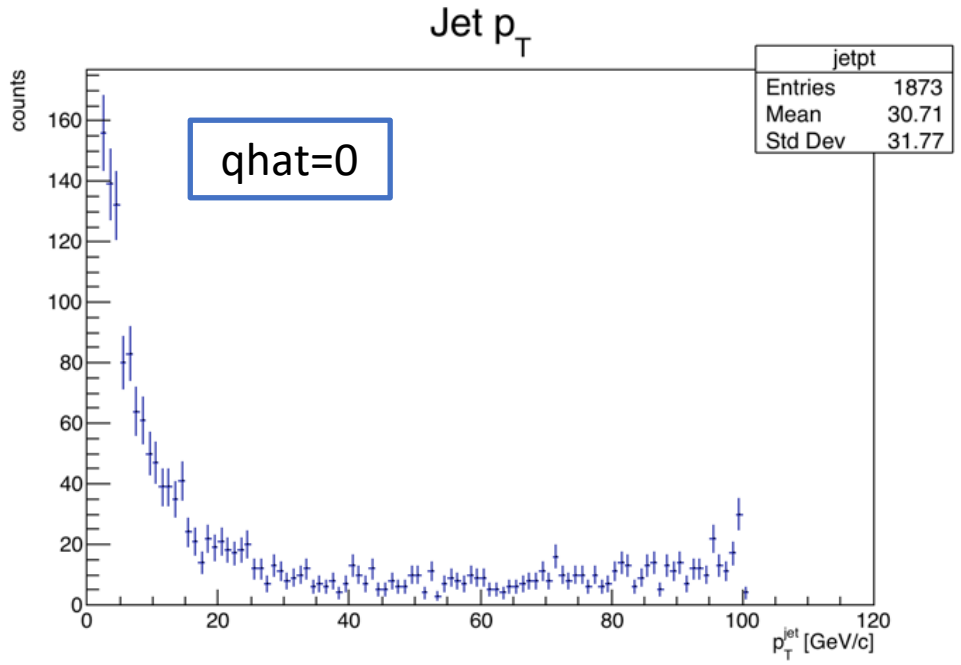
   How? Change the xml file
   (Careful, executable uses xml in your build directory which gets overwritten by cmake)
   → Use renamed xml files and slightly more convenient wrappers provided at
   https://sites.google.com/a/lbl.gov/jetscape2018/home/school-material
3. Create ROOT file

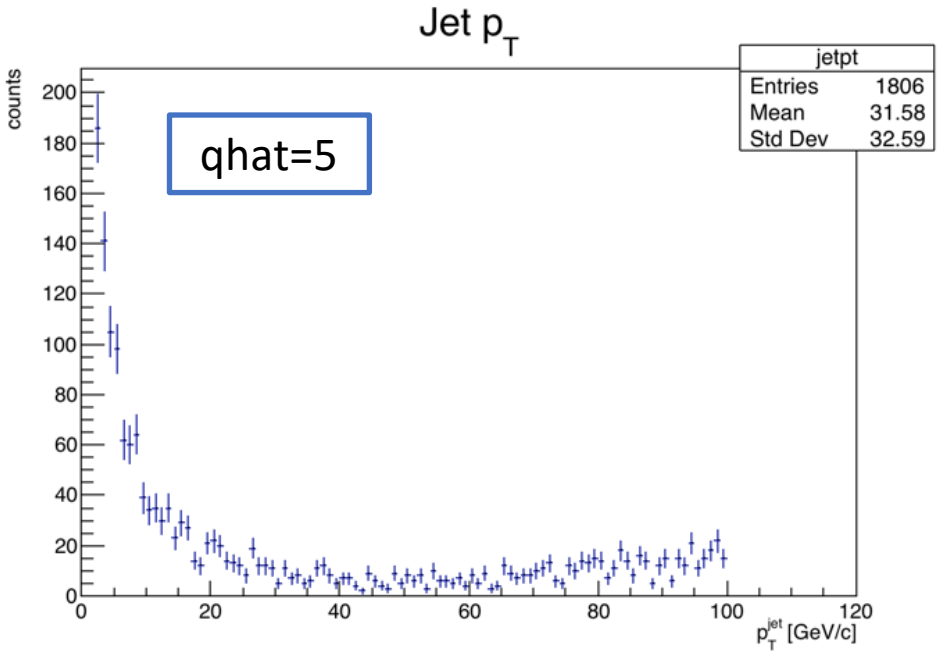
```
bash-3.2$
bash-3.2$ ./brickTest jetscape_init_q0.xml q0.dat 300
bash-3.2$ ./brickTest jetscape_init_q5.xml q5.dat 300
bash-3.2$
bash-3.2$  ./readerTestWithRoot q0.dat
bash-3.2$  ./readerTestWithRoot q5.dat
```

4. Compare jet spectra!

Compare Jet Spectra

```
[bash-3.2$ root -l q0.root
 root [0]
 Attaching file q0.root as _file0...
 (TFile *) 0x7fdcbc149ba0
[root [1] jetpt->Draw()
 Info in <TCanvas::MakeDefCanvas>:  created default TCanvas with name c1
```





No difference? Why?!

Can you think of a better observable?

Addendum

- Could make MATTER thermalize particles like this:

```
// DUMMY THERMALIZATION
// Before processing, catch particles that are absorbed by the medium
// Their virtuality is less than the qhat * the remaining path length
double r = sqrt ( pow( pIn[i].x_in().x(), 2) + pow( pIn[i].x_in().y(), 2) + pow( pIn[i].x_in().z(), 2) ) * fmToGeVinv; // in GeV^-1
double l=length - r;
if ( pIn[i].t() < qhat*l ) {
  VERBOSE(1)  << "Removing particle with virtuality " << pIn[i].t()
              << " at a remaining length of " << l / fmToGeVinv << " fm ";

  // REMOVE PARTICLE FROM pIn
  continue;
}
```

- But the removal needs code restructuring – outside this demo's scope