Analysis prototyping, preservation, and recasting with Rivet

Andy Buckley, University of Glasgow

Corfu HEP & Cosmo Tools Workshop, 10 Sept 2017

Introduction

- **Experiment/theory interaction growing**
  - ⇒ more direct collaboration on methods and modelling, from SM QCD & Top to Higgs and BSM

Introduction

- **Experiment/theory interaction growing**
  - ⇒ more direct collaboration on methods and modelling, from SM QCD & Top to Higgs and BSM
- **Rivet** analysis toolkit is a common dialect for exchanging analysis details and ideas

Introduction

- **Experiment/theory interaction growing**
  - $\Rightarrow$ more direct collaboration on methods and modelling, from SM QCD & Top to Higgs and BSM
- **Rivet** analysis toolkit is a common dialect for exchanging analysis details and ideas
- Implementing a Rivet analysis to complement the data analysis is increasingly expected of ATLAS (and other expt) analyses. **Everyone benefits.**

Introduction

- **Experiment/theory interaction growing**
  - ⇒ more direct collaboration on methods and modelling, from SM QCD & Top to Higgs and BSM
- **Rivet** analysis toolkit is a common dialect for exchanging analysis details and ideas
- Implementing a Rivet analysis to complement the data analysis is increasingly expected of ATLAS (and other expt) analyses. **Everyone benefits.**
- **This talk: description/discussion + demo/exercises**
  Philosophy and recent/relevant developments, plus a few technicalities
  Time limited so I'll skip a lot, but the full set of slides is a useful reference

Rivet

Rivet is an analysis system for MC events, and *lots* of analyses

- Easy and powerful way to get physics numbers & plots from any MC gen

Rivet

Rivet is an analysis system for MC events, and *lots* of analyses

- Easy and powerful way to get physics numbers & plots from any MC gen
  - Only requirement: use **HepMC** event record

Rivet

Rivet is an analysis system for MC events, and *lots* of analyses

- Easy and powerful way to get physics numbers & plots from any MC gen
  - Only requirement: use **HepMC** event record
  - Intentionally unaware of who made the event ⇒ don't "look inside" the event graph

Rivet

Rivet is an analysis system for MC events, and *lots* of analyses

- Easy and powerful way to get physics numbers & plots from any MC gen
  - Only requirement: use **HepMC** event record
  - Intentionally unaware of who made the event ⇒ don't "look inside" the event graph
  - "If you can't write a Rivet analysis for it, it's probably unphysical"!

Rivet

Rivet is an analysis system for MC events, and *lots* of analyses

- Easy and powerful way to get physics numbers & plots from any MC gen
  - Only requirement: use **HepMC** event record
  - Intentionally unaware of who made the event ⇒ don't "look inside" the event graph
  - "If you can't write a Rivet analysis for it, it's probably unphysical"!
- LHC standard for preserving data analyses
  - Key input to MC validation and tuning, and rising relevance to BSM recasting
  - **Add your analyses, too!**

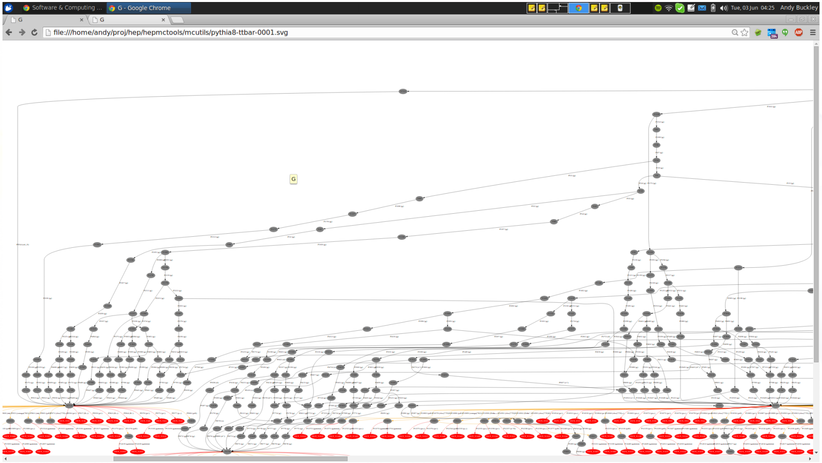
Rivet

Rivet is an analysis system for MC events, and *lots* of analyses

- Easy and powerful way to get physics numbers & plots from any MC gen
  - Only requirement: use **HepMC** event record
  - Intentionally unaware of who made the event ⇒ don't "look inside" the event graph
  - "If you can't write a Rivet analysis for it, it's probably unphysical"!
- LHC standard for preserving data analyses
  - Key input to MC validation and tuning, and rising relevance to BSM recasting
  - **Add your analyses, too!**
- Technical details:
  - C++ library with Python interface & scripts
  - Analyses are "plugins": no need to rebuild
  - Clean interface for ease & expressiveness; efficiency tricks under the hood

Why wouldn't we want to look at the event graph?!

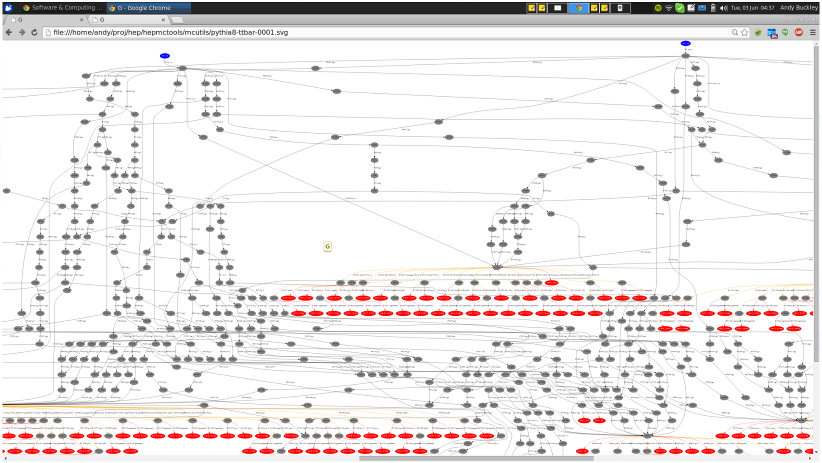
A Pythia8 $t\bar{t}$ event!



Most of this is not standardised: Herwig and Sherpa look *very* different.
But final states and decay chains have to have equivalent meaning.

Why wouldn't we want to look at the event graph?!

A Pythia8 $t\bar{t}$ event!



Most of this is not standardised: Herwig and Sherpa look *very* different.
But final states and decay chains have to have equivalent meaning.

Rivet setup

Local install

Easy to install using our *bootstrap script*:

```
wget http://rivet.hepforge.org/hg/bootstrap/raw-file/2.5.4/rivet-bootstrap
bash rivet-bootstrap
```

Latest version is 2.5.4. Requires C++11

Run from LCG

Can also pick up latest from Genser/LCG build area:

```
ssh lxplus7.cern.ch
. /cvmfs/sft.cern.ch/lcg/releases/LCG_87/gcc/6.2.0/x86_64-centos7/setup.sh
. /cvmfs/sft.cern.ch/lcg/releases/LCG_87/MCGenerators/rivet/2.5.4/...
   x86_64-centos7-gcc62-opt/rivetenv.sh
```

First Rivet runs

Running Rivet

- `rivet` command line tool to query available analyses
- Can be used as a library (e.g. in big experiment software frameworks)
- Can also be used from the command line to read HepMC ASCII files/pipes: very convenient
- Helper scripts like `rivet-mkanalysis`, `rivet-buildplugin`
- Histogram comparisons, plot web albums, etc. very easy

Docs online at http://rivet.hepforge.org – PDF manual, HTML list of existing analyses, and Doxygen.

Viewing available analyses

Rivet knows all sorts of details about its analyses:

- List available analyses:
  `rivet --list-analyses`
- List ATLAS analyses:
  `rivet --list-analyses ATLAS_`
- Show some pure-MC analyses' full details:
  `rivet --show-analysis MC_`

The PDF and HTML documentation is also built from this info, so is always synchronised.

The analysis metadata is provided via the analysis API and usually read from a `.info` file which accompanies the analysis.

Running a simple analysis

To avoid huge files, we get the events from generator to Rivet by writing to a filesystem pipe: `mkfifo fifo.hepmc`
You can also just use a file but it'll be *big*.

NB. A FIFO has to live in a non-AFS dir, e.g. `mkfifo /tmp/$USER/fifo.hepmc`

Running a simple analysis

To avoid huge files, we get the events from generator to Rivet by writing to a filesystem pipe: `mkfifo fifo.hepmc`
You can also just use a file but it'll be *big*.

NB. A FIFO has to live in a non-AFS dir, e.g. `mkfifo /tmp/$USER/fifo.hepmc`

I'm going to use the Sacrifice frontend to run Pythia 8 for demonstration – use the same or run any other generator that you like with HepMC output going to the FIFO:

```
run-pythia -n 2000 -c Top:all=on -o fifo.hepmc &
```

Running a simple analysis

To avoid huge files, we get the events from generator to Rivet by writing to a filesystem pipe: `mkfifo fifo.hepmc`
You can also just use a file but it'll be *big*.

NB. A FIFO has to live in a non-AFS dir, e.g. `mkfifo /tmp/$USER/fifo.hepmc`

I'm going to use the Sacrifice frontend to run Pythia 8 for demonstration – use the same or run any other generator that you like with HepMC output going to the FIFO:

```
run-pythia -n 2000 -c Top:all=on -o fifo.hepmc &
```

Now attach Rivet to the other end of the pipe:

```
rivet -a MC_GENERIC -a MC_JETS fifo.hepmc
```

Running a simple analysis

To avoid huge files, we get the events from generator to Rivet by writing to a filesystem pipe: `mkfifo fifo.hepmc`
You can also just use a file but it'll be *big*.
NB. A FIFO has to live in a non-AFS dir, e.g. `mkfifo /tmp/$USER/fifo.hepmc`

I'm going to use the Sacrifice frontend to run Pythia 8 for demonstration – use the same or run any other generator that you like with HepMC output going to the FIFO:

```
run-pythia -n 2000 -c Top:all=on -o fifo.hepmc &
```

Now attach Rivet to the other end of the pipe:

```
rivet -a MC_GENERIC -a MC_JETS fifo.hepmc
```

Hopefully that worked. You can use multiple analyses at once, change the output file, etc.: see `rivet --help`

By default *unfinalised* histos are written every 1000 events: can monitor progress through the run. Killing with `Ctrl-C` is safe: finalizing is run

Example output

```
$ run-pythia -e 7000 -c HardQCD:all=on -c ParticleDecays:limitTau0=on
    -n 10000 -o fifo.hepmc &
$ rivet -a CMS_2013_I1265659 fifo.hepmc
$ rivet-mkhtml -a Rivet.yoda:'Py8 $\star$'
```

```
# BEGIN YODA_HISTO1D /CMS_2013_I1265659/d01-x01-y02
Path=/CMS_2013_I1265659/d01-x01-y02
ScaledBy=0.00018488029661016948
Title=
Type=Histo1D
XLabel=
YLabel=
# Mean: 1.886500e+00
# Area: 1.745270e-01
# xlow   xhigh   sumw    sumw2   sumwx   sumwx2  numEntries
Total           Total           1.745270e-01    3.226660e-05    3.292452e-01    7.563865e-01    944
Underflow       Underflow       0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00    0
Overflow        Overflow        0.000000e+00    0.000000e+00    0.000000e+00    0.000000e+00    0
1.001800e-04    1.746272e-01    4.622007e-03    8.545181e-07    3.464255e-04    3.868572e-05    25
1.746276e-01    3.491546e-01    6.101050e-03    1.127964e-06    1.634274e-03    4.481578e-04    33
3.491549e-01    5.236819e-01    6.840571e-03    1.264687e-06    2.938932e-03    1.279250e-03    37
5.236823e-01    6.982093e-01    7.395212e-03    1.367229e-06    4.569311e-03    2.838956e-03    40
6.982097e-01    8.727367e-01    6.285930e-03    1.162145e-06    4.880735e-03    3.805391e-03    34
8.727370e-01    1.047264e+00    6.470810e-03    1.196325e-06    6.237378e-03    6.024974e-03    35
1.047265e+00    1.221791e+00    7.395212e-03    1.367229e-06    8.247895e-03    9.216318e-03    40
                                                .
                                                .
                                                .
# END YODA_HISTO1D
```

Plotting histograms

ROOT didn’t meet our requirements :-(

bin width issues, bin gaps unhandled, object ownership nightmare, thread-unsafety

Rivet uses custom “YODA” stats library – http://yoda.hepforge.org

Plotting histograms

ROOT didn't meet our requirements :-(

bin width issues, bin gaps unhandled, object ownership nightmare, thread-unsafety

Rivet uses custom "YODA" stats library – http://yoda.hepforge.org

- YODA data format is plain text and *stores all second-order statistical moments*: can do full stat merging, compute all means and variances
- Plus general metadata annotation system – styling, notes, whatever – and evolution of data types optimised for MC

CLI tools: **`yodals`**, **`yodadiff`**, **`yodamerge`**, **`yodascale`**, **`yoda2root`**, etc.

Plotting histograms

ROOT didn't meet our requirements :-(

bin width issues, bin gaps unhandled, object ownership nightmare, thread-unsafety

Rivet uses custom "YODA" stats library – http://yoda.hepforge.org

- YODA data format is plain text and *stores all second-order statistical moments*: can do full stat merging, compute all means and variances
- Plus general metadata annotation system – styling, notes, whatever – and evolution of data types optimised for MC

CLI tools: `yodals`, `yodadiff`, `yodamerge`, `yodascale`, `yoda2root`, etc.

Plotting a `.yoda` file is easy: `rivet-mkhtml Rivet.yoda`

Advanced: `rivet-mkhtml Rivet.yoda:'Pythia\,8 $t\bar{t}$'`

or, if you want complete control:

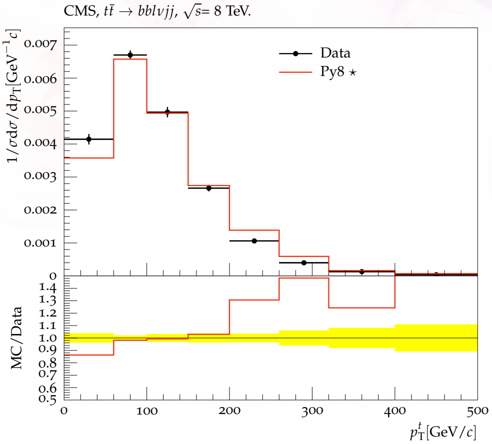
```
rivet-cmphistos Rivet.yoda:'My title':LineColor=red && make-plots *.dat
```

Then view with a web browser/file browser/evince/...

NB. A `--help` option is available for all Rivet scripts.

Example output

```
zcat tt-20k.yoda | rivet -a
ATLAS_2015_I1376945,CMS_2015_I1370682,CMS_2016_I1473674
$ rivet-mkhtml -a Rivet.yoda:'Py8 $\star$'
```



Writing a first analysis

Writing an analysis

Writing an analysis is of course more involved
But the C++ interface is pretty friendly: most analyses are short, simple, and readable

An example is usually the best instruction: take a look at
http://rivet.hepforge.org/hg/rivet/file/tip/src/Analyses/MC_GENERIC.cc

Writing an analysis

Writing an analysis is of course more involved
But the C++ interface is pretty friendly: most analyses are short, simple, and readable

An example is usually the best instruction: take a look at
http://rivet.hepforge.org/hg/rivet/file/tip/src/Analyses/MC_GENERIC.cc

Code is "mostly normal":

- Typical init/exec/fin loop structure
- Histograms ∼normal; titles, etc. → external **`.plot`** file
- Particle, Jet and FourMomentum classes with some nice things like **`abseta()`** and **`abspid()`**, constituents, decay-chain searching, and compatibility with FastJet objects
- Use of *projections* for auto-cached computations

Projections

Projections are just observable calculators: given an `Event` object, they *project* out physical observables.

Automatic caching of results leads to slightly odd calling code:

Declaration with a string name in the `init` method:

```
void init() {
  ...
  const SomeProjection sp(foo, bar);
  declare(sp, "MySP");
  ...
}
```

Application in the `analyze` method via the same name:

```
void analyze(const Event& evt) {
  ...
  const SomeProjectionBase& mysp =
    apply<SomeProjectionBase>(evt, "MySP");
  mysp.foo()
  ...
}
```

Then query it about the things it has computed, via the object/ref API

Particle finders & final-state projections

Rivet is mildly obsessive about calculating from final state objects

So a *very* important set of projections is those used to extract final state particles, which inherit from `FinalState`

- The `FinalState` projection finds all final state particles in a given η range, with a given p_T cutoff.
- Subclasses `ChargedFinalState` and `NeutralFinalState` have the predictable effect!
- `IdentifiedFinalState` can be used to find particular particle species. Nowadays arguably done more nicely via a `Cut`
- `VetoedFinalState` finds particles *other* than specified. Ditto
- `VisibleFinalState` excludes invisible particles like neutrinos, LSP

NB. Most FSPs can take another FSP as a constructor argument and augment it

Using an FSP to get final state particles

```
void init() {
  ...
  const ChargedFinalState cfs(Cuts::pT > 500*MeV && Cuts::abseta < 2.5);
  declare(cfs, "ChFS");
  ...
}
```

```
void analyze(const Event& evt) {
  ...
  const FinalState& cfs =
    apply<FinalState>(evt, "ChFS");
  MSG_INFO("Total charged mult. = " << cfs.size());
  for (const Particle& p : cfs.particles()) {
    MSG_DEBUG("Particle eta = " << p.eta());
  }
  ...
}
```

More complex projections like **`DressedLeptons`**, **`FastJets`**, **`WFinder`**, **`TauFinder`** …implement expt-like strategies for dressing, tagging, mass-windowing, etc.

Selection cuts

Passing ordered lists of doubles to configure "automatic" cut rules is inflexible, illegible, and error-prone. So…

Combinable `Cut` objects:

- `FinalState(Cuts::pT > 0.5*GeV && Cuts::abseta < 2.5)`
- `fs.particles(Cuts::absrap < 3 || (Cuts::absrap > 3.2 && Cuts::absrap < 5), cmpMomByEta)`

Can also use cuts on PID and charge:

- `fs.particlesByPt(Cuts::abspid == PID::ELECTRON)`, or
- `FinalState(Cuts::charge != 0)`

Use of *functions/functors* for ParticleFinder filtering is also possible: very general, especially with C++ *lambdas*

Jets

One more important projection set is those which find *jets*

There's a `JetAlg` abstract interface, but almost always use FastJet, via `FastJets`

Define the input particles (via a `FinalState`), and the jet alg & params:

```
const FinalState fs(-3.2, 3.2);
declare(fs, "FS");
FastJets fj(fs, FastJets::ANTIKT, 0.6,
            JetAlg::ALL_MUONS, JetAlg::ALL_INVISIBLES);
declare(fj, "Jets");
```

Get the jets and loop over them in decreasing p_T order:

```
const Jets jets =
  apply<JetAlg>(evt, "Jets").jetsByPt(20*GeV);
for (const Jet& j : jets) {
  for (const Particle& p : j.particles()) {
    const double dr = deltaR(j, p); //< auto-conversion!
  }
}
```

Remember to `#include "Rivet/Projections/FastJets.hh"`

NB. Lots of handy functions in `Rivet/Math/MathUtils.hh`!

Jet tagging

Jet flavour tagging can use a very inclusive tagging definition based on hadron parentage, without requiring kinematic closeness to the jet:

- `j.hasBottom()`

Still an option, but now also automatically ghost-tag jets using *b* and *c* hadrons:

- `if (!myjet.bTags().empty()) ...`

And you can use Cuts to refine the truth tag:

- `myjet.bTags(Cuts::abseta < 2.5 && Cuts::pT > 5*GeV)`

Jet substructure

Looking inside jets is common practice these days!

Rivet doesn't duplicate existing tools: best just to use FastJet directly

```
const PseudoJets psjets = fj.pseudoJets();
const ClusterSequence* cseq = fj.clusterSeq();

Selector sel_3hardest = SelectorNHardest(3);
Filter filter(0.3, sel_3hardest);
for (const PseudoJet& pjet : psjets) {
  PseudoJet fjet = filter(pjet);
  ...
}
```

Note: if using FastJet3 tools, you'll need to add `lifastjettools` to the `rivet-buildplugin` command line. And a `-L/path/to/` arg as well, until the next release. Just compilation, no magic

Rivet's `Jet` and `Particle` classes auto-convert to `PseudoJet`:
⇒ `d23 = cs.exclusive_subdmerge(jetproj.jetsByPt[0], 2)`

Writing, building & running your own analysis

Let's start with a simple "particle analysis", just plotting some simple particle properties like η, p_T, ϕ, etc. Then we'll try jets or W/Z.

To get an analysis template, which you can fill in with an FS projection and a particle loop, run e.g. `rivet-mkanalysis MY_TEST_ANALYSIS` – this will make the required files.

Once you've filled it in, you can either compile directly with `g++`, using the `rivet-config` script as a compile flag helper, or run `rivet-buildplugin MY_TEST_ANALYSIS.cc`

To run, first `export RIVET_ANALYSIS_PATH=$PWD`, then run `rivet` as before… or add the `--pwd` option to the `rivet` command line.

BSM searches and detector effects

BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies

MC truth

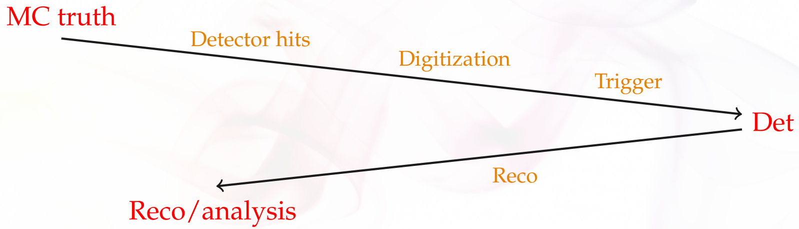
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies

$$\begin{tikzcd}\text{MC truth} \arrow[r, "\text{Detector hits} \quad \text{Digitization} \quad \text{Trigger}"] & \text{Det}\end{tikzcd}$$

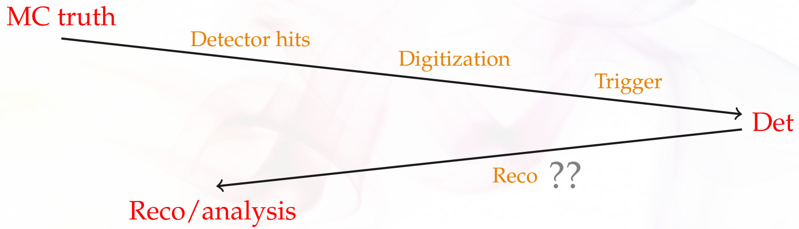
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



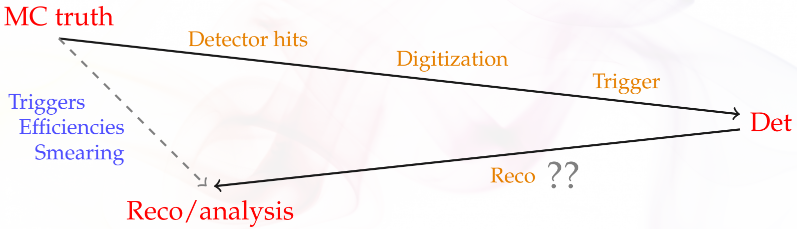
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



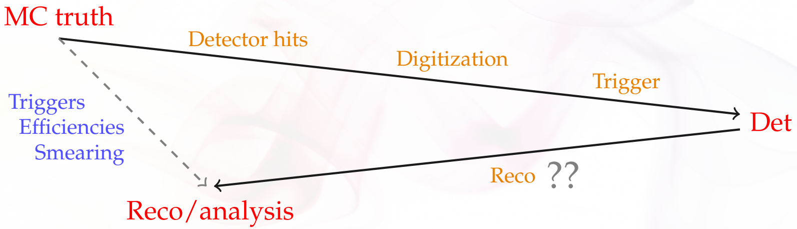
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



- **Explicit fast-sim takes the "long way round".**
- **Reco already reverses most detector effects!**
- Reco calibration to MC truth: smearing is a few-percent effect
- (Lepton) efficiency & mis-ID functions dominate – and are tabulated in both approaches
- Smearing is more flexible: effs change with phase-space, reco version, run, . . . and need to guarantee *stability* for preservation

Detector effects in Rivet

In addition to last slides, *flexibility* of det-sim is important:

- "Global" fast-sims hence difficult for coverage of multiple experiments, multiple runs, multiple reco calibrations, etc.
- Analysis-specific efficiencies and smearings are more precise and allow use of multiple jet sizes, tagger & ID working points, isolations, . . . ⇒ **many variations in real analyses**

⇒ Rivet det-sim as effs+smearing, localised per-analysis

Rivet internally caches results, so global effect sim still efficient

- Functions for generic ATLAS & CMS performance in Runs 1 & 2
- Inline or analysis-specific functions easy to write & *chain*
- Eff/smearing functions can be used directly, e.g. for object filtering
- Working on embeddability for multithreaded fitters/samplers.

Using Rivet's fast-sim tools

Smearing is provided as "wrapper projections" on normal particle, jet, and MET finders. Maximal flexibility and minimal impact on unfolded analysis tools. Smearing configuration via efficiency/modifier functions.

To use, first `#include "Rivet/Projections/Smearing.hh"`

Examples:

```
IdentifiedFinalState es1(Cuts::abseta < 5, {{PID::ELECTRON, PID::POSITRON}});
SmearedParticles es2(es, ELECTRON_EFF_ATLAS_RUN2, ELECTRON_SMEAR_ATLAS_RUN2);
declare(recoes, "Electrons");

FastJets js1(FastJets::ANTIKT, 0.6, JetAlg::DECAY_MUONS);
SmearedJets js2(fj, JET_SMEAR_PERFECT, JET_EFF_BTAG_ATLAS_RUN2); // or lambda
declare(recoj, "Jets");

...

Particles elecs = apply<ParticleFinder>(event, "Electrons").particles(10*GeV);
Jets jets = apply<JetAlg>(event, "Jets").jetsByPt(30*GeV);
```

Note set of standard global functions. Private fns also ok. *Inline* via *C++11 lambda fns*

Small tweak planned, to unify eff/mod fns and give user control of *operator ordering*

Selection tools for search analyses

Search analyses typically do a lot more "object filtering" than measurements. Rivet provides a lot of tools to make this complex logic expressive:

- Filtering functions: `filter_select(const Particles/Jets&, FN)`, `filter_discard(...)` + `ifilter_*` in-place variants
- Lots of *functors* for common "stateful" filtering criteria: `PtGtr(10*GeV)`, `EtaLess(5)`, `AbsEtaGtr(2.5)`, `DeltaRGtr(mom, 0.4)`
    - Lots of these in `Rivet/Tools/ParticleBaseUtils.hh`, `Rivet/Tools/ParticleUtils.hh`, and `Rivet/Tools/JetUtils.hh`
- `any()`, `all()`, `none()`, etc. – accepting functions/functors
- Cut-flow monitor via `#include "Rivet/Tools/Cutflow.hh"`

Selection tools: examples

```
const Jets jets = apply<JetAlg>(event, "Jets")
  .jetsByPt(Cuts::pT > 20*GeV && Cuts::abseta < 2.8);
const Particles elecs = apply<ParticleFinder>(event, "Elecs").particlesByPt();
const Particles mus = apply<ParticleFinder>(event, "Muons").particlesByPt();
MSG_DEBUG("Number of raw jets, electrons, muons = "
          << jets.size() << ", " << elecs.size() << ", " << mus.size());
```

Selection tools: examples

```
const Jets jets = apply<JetAlg>(event, "Jets")
  .jetsByPt(Cuts::pT > 20*GeV && Cuts::abseta < 2.8);
const Particles elecs = apply<ParticleFinder>(event, "Elecs").particlesByPt();
const Particles mus = apply<ParticleFinder>(event, "Muons").particlesByPt();
MSG_DEBUG("Number of raw jets, electrons, muons = "
          << jets.size() << ", " << elecs.size() << ", " << mus.size());
```

```
// Discard jets very close to electrons, or low-ntrk jets close to muons
const Jets isojets = filter_discard(jets, [&](const Jet& j) {
    if (any(elecs, deltaRLess(j, 0.2))) return true;
    if (j.particles(Cuts::abscharge > 0 && Cuts::pT > 0.4*GeV).size() < 3 &&
        any(mus, deltaRLess(j, 0.4))) return true;
    return false;
  });
```

Selection tools: examples

```
const Jets jets = apply<JetAlg>(event, "Jets")
  .jetsByPt(Cuts::pT > 20*GeV && Cuts::abseta < 2.8);
const Particles elecs = apply<ParticleFinder>(event, "Elecs").particlesByPt();
const Particles mus = apply<ParticleFinder>(event, "Muons").particlesByPt();
MSG_DEBUG("Number of raw jets, electrons, muons = "
          << jets.size() << ", " << elecs.size() << ", " << mus.size());
```

```
// Discard jets very close to electrons, or low-ntrk jets close to muons
const Jets isojets = filter_discard(jets, [&](const Jet& j) {
    if (any(elecs, deltaRLess(j, 0.2))) return true;
    if (j.particles(Cuts::abscharge > 0 && Cuts::pT > 0.4*GeV).size() < 3 &&
        any(mus, deltaRLess(j, 0.4))) return true;
    return false;
  });
```

```
// Discard electrons close to remaining jets
const Particles isoelecs = filter_discard(elecs, [&](const Particle& e) {
    return any(isojets, deltaRLess(e, 0.4));
  });
```

Selection tools: examples

```
const Jets jets = apply<JetAlg>(event, "Jets")
  .jetsByPt(Cuts::pT > 20*GeV && Cuts::abseta < 2.8);
const Particles elecs = apply<ParticleFinder>(event, "Elecs").particlesByPt();
const Particles mus = apply<ParticleFinder>(event, "Muons").particlesByPt();
MSG_DEBUG("Number of raw jets, electrons, muons = "
          << jets.size() << ", " << elecs.size() << ", " << mus.size());
```

```
// Discard jets very close to electrons, or low-ntrk jets close to muons
const Jets isojets = filter_discard(jets, [&](const Jet& j) {
    if (any(elecs, deltaRLess(j, 0.2))) return true;
    if (j.particles(Cuts::abscharge > 0 && Cuts::pT > 0.4*GeV).size() < 3 &&
        any(mus, deltaRLess(j, 0.4))) return true;
    return false;
  });
```

```
// Discard electrons close to remaining jets
const Particles isoelecs = filter_discard(elecs, [&](const Particle& e) {
    return any(isojets, deltaRLess(e, 0.4));
  });
```

...

Selection tools: examples

```
const Jets jets = apply<JetAlg>(event, "Jets")
  .jetsByPt(Cuts::pT > 20*GeV && Cuts::abseta < 2.8);
const Particles elecs = apply<ParticleFinder>(event, "Elecs").particlesByPt();
const Particles mus = apply<ParticleFinder>(event, "Muons").particlesByPt();
MSG_DEBUG("Number of raw jets, electrons, muons = "
          << jets.size() << ", " << elecs.size() << ", " << mus.size());
```

```
// Discard jets very close to electrons, or low-ntrk jets close to muons
const Jets isojets = filter_discard(jets, [&](const Jet& j) {
    if (any(elecs, deltaRLess(j, 0.2))) return true;
    if (j.particles(Cuts::abscharge > 0 && Cuts::pT > 0.4*GeV).size() < 3 &&
        any(mus, deltaRLess(j, 0.4))) return true;
    return false;
  });
```

```
// Discard electrons close to remaining jets
const Particles isoelecs = filter_discard(elecs, [&](const Particle& e) {
    return any(isojets, deltaRLess(e, 0.4));
  });
```

...

```
zcat susy-gg-g1000-chi800-g2qqchi-20k.hepmc.gz | rivet -a
CMS_2017_I1594909 -H out.yoda
```

That's all, folks

Summary

- **Rivet is a user-friendly MC analysis system for prototyping and preserving data analyses**
- Allows theorists to use your analyses for model development & testing, and BSM recasting: impact beyond "get a paper out"
- Also a very useful cross-check: quite a few analysis bugs have been found via Rivet!
- Strongly encouraged/required by ATLAS & CMS physics groups. Integrated with experiment software
- Now supports detector simulation for BSM search preservation
- Multi-weights, NLO counter-events, and multi-threading all in the pipeline
- **Feedback, questions and getting involved in development all very welcome!**

Backup

Running a data analysis

For example, the ATLAS 7 TeV high-p_T jet shapes analysis:

```
rivet --show-analysis ATLAS_2012_I1119557
```

Note: tab completion for `rivet` options and analysis names.

Running a data analysis

For example, the ATLAS 7 TeV high-p_T jet shapes analysis:

```
rivet --show-analysis ATLAS_2012_I1119557
```

Note: tab completion for `rivet` options and analysis names.

Now to run it:

```
run-pythia -n 20000 -e 7000 -c HardQCD:all=on -c
PhaseSpace:pTHatMin=280 -o fifo.hepmc &
```

```
rivet -a ATLAS_2012_I1119557 fifo.hepmc
```

See the Py8 manual: http://home.thep.lu.se/∼torbjorn/pythia82html/Welcome.html

Running a data analysis

For example, the ATLAS 7 TeV high-p_T jet shapes analysis:

```
rivet --show-analysis ATLAS_2012_I1119557
```

Note: tab completion for `rivet` options and analysis names.

Now to run it:

```
run-pythia -n 20000 -e 7000 -c HardQCD:all=on -c
PhaseSpace:pTHatMin=280 -o fifo.hepmc &

rivet -a ATLAS_2012_I1119557 fifo.hepmc
```

See the Py8 manual: http://home.thep.lu.se/~torbjorn/pythia82html/Welcome.html

And plot, much as before:

```
rivet-mkhtml Rivet.yoda:Pythia8
```

By default *unfinalised* histos are written every 1000 events: can monitor progress through the run. Killing with `Ctrl-C` is safe: finalizing is run

Feeding LHEF events into Rivet

If your code outputs LHEF events rather than HepMC, you can't use Rivet directly. Anyway, you're taking a risk that it won't work since Rivet is final-state focused…but you can also get hold of the raw event if you want and just use the histogramming and event loop.

At Les Houches 2011 I made a mini filter program which will convert LHEF files or streams to HepMC ones:
http://rivet.hepforge.org/hg/contrib/file/tip/lhef2hepmc/

Use it like this:

```
./lhef2hepmc fifo.lhef fifo.hepmc
```

or

```
./lhef2hepmc fifo.lhef - | rivet
```

Maybe some help will be needed with building this program – it's not an official part of Rivet so you have to download and build it by hand. Let us know if you need a hand.

Running Rivet in Athena

Rivet is interfaced to the ATLAS Athena framework: see https://twiki.cern.ch/twiki/bin/viewauth/AtlasProtected/RivetForAtlas for all sorts of guidance

Basic setup:

```
setupATLAS
lsetup asetup
asetup 20.7.9.9
rivet --version
```

another way to get command-line Rivet

For running in vanilla **athena**:

```
get_files -jo jobOptions.rivet.py
athena jobOptions.rivet.py
```

and edit

Or built-in to running ATLAS generators:

```
Generate_tf.py ... --rivetAnas=MC_GENERIC,MC_JETS ...
```

More about Rivet/YODA histogramming & merging

- **YODA allows "simple" automatic run merging.** With some heuristics to distinguish homogeneous and heterogeneous run types.
- **Not complete:** merging (normalised) histograms and profiles is one thing, but what about general objects, particularly ratios like H_A/H_B (or more complex)
- **YODA paves the way to a complete treatment:**
  - User-accessible histograms will only be temporary copies for the current event group (to allow weight vectors & counter-events)
  - Synchronised to a less transient copy every time the event number changes in the event loop
  - Periodically, or on **`finalize()`**, this second copy gets used to make *final* histograms: normalised, scaled, added, etc.
  - "Final" histograms can be written and updated through the run: **`finalize()`** runs many times
  - And runs can be re-loaded and combined using the pre-finalize copies ⇒ completely general run combination.
- Also tie-in with heavy ion / process-ratio analysis workflow

Projections – registration

Major idea: **projections**. They are just observable calculators: given an `Event` object, they *project* out physical observables.

They also automatically cache themselves, to avoid recomputation. This leads to slightly unfamiliar calling code.

They are *declared* with a name in the `init` method:

```
void init() {
  ...
  const SomeProjection sp(foo, bar);
  declare(sp, "MySP");
  ...
}
```

Projections – applying

Projections were declared with a name…they are then *applied* to the current event, also by name:

```
void analyze(const Event& evt) {
   ...
   const SomeProjectionBase& mysp =
     apply<SomeProjectionBase>(evt, "MySP");
   mysp.foo()
   ...
}
```

We prefer to get a handle to the applied projection as a const reference to avoid unnecessary copying.

It can then be queried about the things it has computed. Projections have different abilities and interfaces: check the Doxygen on the Rivet website, e.g. **http://projects.hepforge.org/rivet/code/dev/hierarchy.html**

Physics vectors

Rivet uses its own physics vectors rather than CLHEP or ROOT. They are a little nicer to use (we think!), but basically familiar. As usual, check Doxygen: **http://projects.hepforge.org/rivet/code/dev/**

`Particle` and `Jet` both have a `momentum()` method which returns a `FourMomentum`.

Some `FourMomentum` methods: `eta()`, `pT()`, `phi()`, `rapidity()`, `E()`, `px()` etc., `mass()`. Hopefully intuitive!

Histogramming

YODA has Histo1D and Profile1D histograms (and more), which behave as you would expect. See
http://yoda.hepforge.org/doxy/hierarchy.html

Histogramming

YODA has Histo1D and Profile1D histograms (and more), which behave as you would expect. See
http://yoda.hepforge.org/doxy/hierarchy.html

Histos are booked via helper methods on the `Analysis` base class, which deal with path issues and some other abstractions*: e.g.
`bookHisto1D("thisname", 50, 0, 100)`
Histo binnings can also be booked via a vector of bin edges or *autobooked* from a reference histogram.

Histogramming

YODA has Histo1D and Profile1D histograms (and more), which behave as you would expect. See
http://yoda.hepforge.org/doxy/hierarchy.html

Histos are booked via helper methods on the **`Analysis`** base class, which deal with path issues and some other abstractions*: e.g.
`bookHisto1D("thisname", 50, 0, 100)`
Histo binnings can also be booked via a vector of bin edges or *autobooked* from a reference histogram.

The histograms have the usual **`fill(value, weight)`** method for use in the **`analyze`** method. There are **`scale()`**, **`normalize()`** and **`integrate()`** methods for use in **`finalize()`**.

Histogramming

YODA has Histo1D and Profile1D histograms (and more), which behave as you would expect. See
http://yoda.hepforge.org/doxy/hierarchy.html

Histos are booked via helper methods on the `Analysis` base class, which deal with path issues and some other abstractions*: e.g.
`bookHisto1D("thisname", 50, 0, 100)`
Histo binnings can also be booked via a vector of bin edges or *autobooked* from a reference histogram.

The histograms have the usual `fill(value, weight)` method for use in the `analyze` method. There are `scale()`, `normalize()` and `integrate()` methods for use in `finalize()`.

The fill weight is important! For kinematic enhancements, systematics, counter-events, etc. Use `evt.weight()` Until automatic multiweight support…

* The abstractions are key to handling systematics weight vectors, correlated counter-events, completely general run merging, etc.

Histogram autobooking

The final framework feature to introduce is histogram autobooking. This is a means for getting your Rivet histograms binned with the same bin edges as used in the experimental data that you'll be comparing to.

To use autobooking, just call the booking helper function with only the histogram name (check that this matches the name in the reference `.yoda` file), e.g.

```
_hist1 = bookHisto1D("d01-x01-y01")
```

The "d", "x" and "y" terms are the indices of the HepData dataset, *x*-axis, and *y*-axis for this histogram in this paper.

A neater form of the helper function is available and should be used for histogram names in this format:

```
_hist1 = bookHisto1D(1, 1, 1)
```

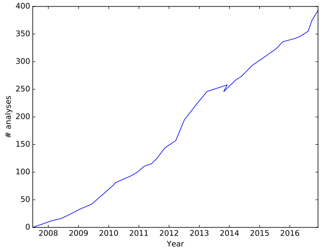
That's it! If you need to get the binnings without booking a persistent histogram use `refData(name)` or `refData(d,x,y)`.

NB. Extra bool argument for using ref data x vals for `Scatter2D`s

BSM analysis coverage

Currently ∼ 427 analyses total & ∼ 230 LHC alone

- Until recently only 27 dedicated BSM searches – and BSM-sensitive SM measurements
- SM focus on unfolded observables, not sufficient for most BSM studies
- Rivet 2.5.0 introduced detector smearing machinery. *For BSM only!*



NB. glitch is Rivet 1.x → 2.x migration.

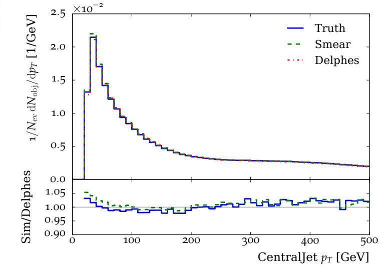
Note recent acceleration!

- ⇒ have coded up 9 more BSM routines in last few months:
  - ATLAS: ICHEP 2016 3-lepton & same-sign 2-lepton, 1-lepton + jets, 1-lepton + many jets, jets + MET; 2015 jets + MET and monojet
  - CMS: ICHEP 2016 jets + MET; 8 TeV α_T + b-jets
  - *Partially* validated – not many cutflows available!
  - Also added tools to help with object filtering, cutflows, etc.
  - Important as real-world examples of how to write BSM routines
- **Rivet is in good shape for preserving new physics searches!**

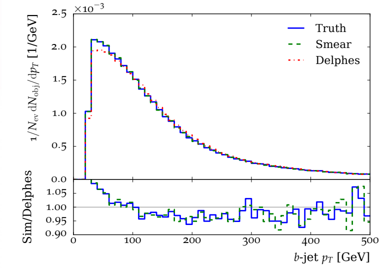
Smearing vs. fast sim vs. MC truth

CMSSM eff/smearing effects from Rivet, in turn using some DELPHES and paper/note calibration functions:

Central jet p_T



b-jet p_T

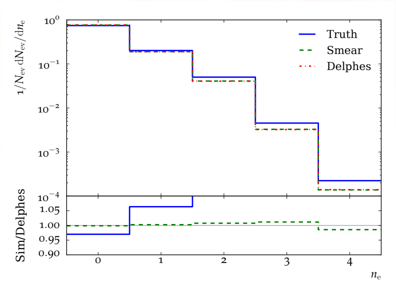


Note major lepton shifts from blue truth to green smeared: difference w.r.t red DELPHES very small

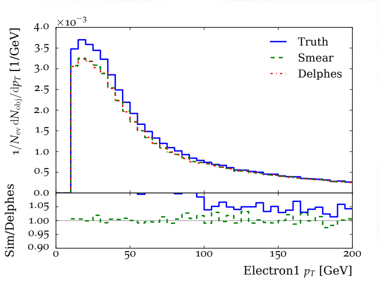
Smearing vs. fast sim vs. MC truth

CMSSM eff/smearing effects from Rivet, in turn using some DELPHES and paper/note calibration functions:

Electron multiplicity



Leading electron p_T

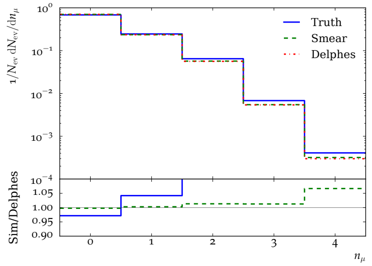


Note major lepton shifts from blue truth to green smeared: difference w.r.t red DELPHES very small

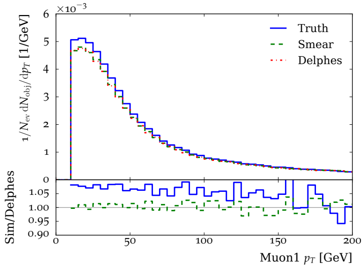
Smearing vs. fast sim vs. MC truth

CMSSM eff/smearing effects from Rivet, in turn using some DELPHES and paper/note calibration functions:

Muon multiplicity



Leading muon p_T



Note major lepton shifts from blue truth to green smeared: difference w.r.t red DELPHES very small