Rivet tutorial — getting started

Holger Schulz
IPPP Durham

SLAC, 11 May 2017

rivet.hepforge.org

Introduction

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

427 built-in, at today's count! 54 are pure MC, and some double/triple-counting

- *Generator-agnostic* for physics & pragmatics
- A quick, easy and powerful way to get physics plots from lots of MC gens
  - Only requirement: use **HepMC** event record
  - Usually via ASCII, but in-memory exchange is faster
- Rivet has become the LHC standard for archiving LHC data analyses
  - Focus on *unfolded* measurements, esp. QCD and EW+QCD, rather than searches
  - But there are BSM studies using it! **And detector simulation now possible**
  - Key input to MC validation and tuning – increasingly comprehensive coverage
  - Also "recasting" of SM and BSM data results on to new/more general BSM model spaces
  - **Add your analyses, too!**

Design philosophy / pragmatics

Rivet operates on HepMC events, intentionally unaware of who made them...so don't "look inside" the event graph.

⇒ reconstruct resonances, dress leptons, avoid partons, etc.

cf. q/g jet discrimination: LO picture is an implementation-dependent cartoon; a useful motivator but incomplete and ill-defined until after hadronization

This "hard work" way is actually simpler – fewer gotchas.
Makes you think about physics & helps find analysis bugs/ambiguities

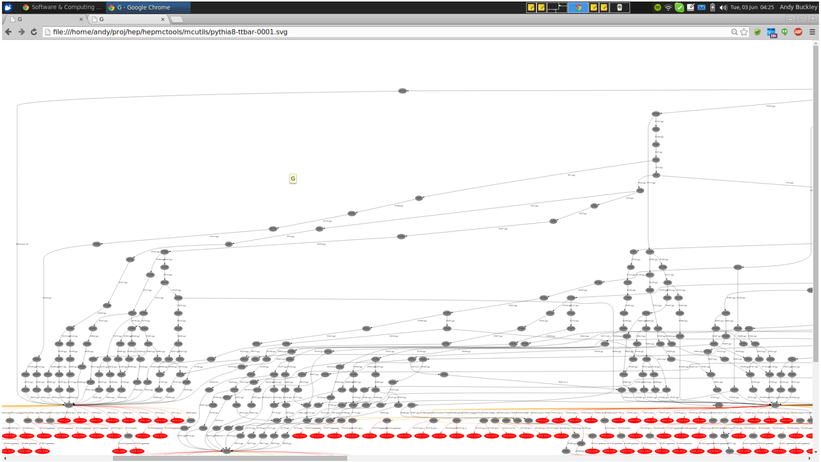
Tech stuff:

- C++ library with Python interface & scripts
- "Plugins" ⇒ write your analyses without needing to rebuild Rivet
  Trivial from user / analysis author point of view
- Tools to make "doing things properly" easy and default
- Computation caching for efficiency
- Histogram syncing: *keep code clean and clear*

+ helpful developers! New contributors always welcome

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

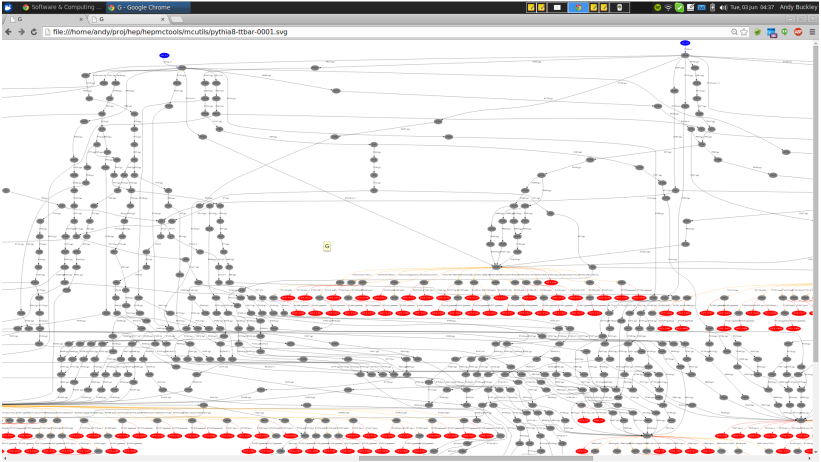
A Pythia8 $t\bar{t}$ event!



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

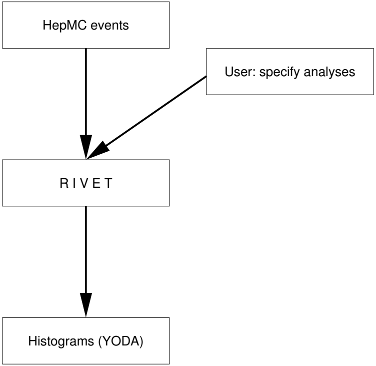
Why wouldn't we want to look at the event graph?!

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

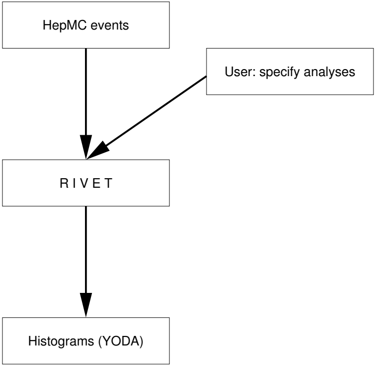


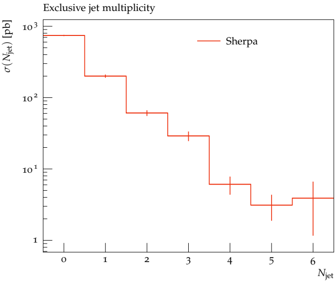
Most of this is not standardised: Herwig and Sherpa look *very* different.
But final states and decay chains have to have equivalent meaning.

Basic principle

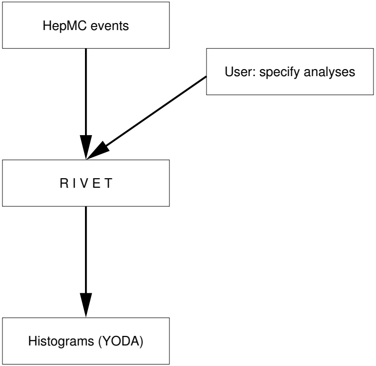


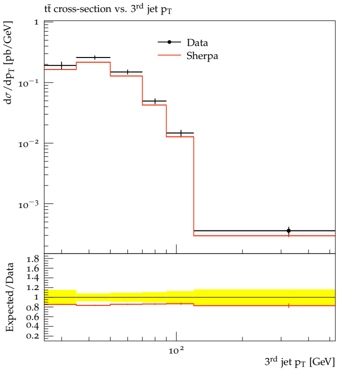
Basic principle





Basic principle





Running Rivet

Rivet is readily available on the MC4BSM virtual machine.
Some event files are available in **`tutorial/Rivet/Events`**

Easy to install using our *bootstrap script*:

```
wget http://rivet.hepforge.org/hg/bootstrap/raw-file/2.5.3/rivet-bootstrap
bash rivet-bootstrap
```

Latest version is 2.5.3 Requires C++11 Docker image available:

```
docker pull hepstore/rivet:2.5.3
```

http://rivet.hepforge.org/trac/wiki/Docker

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. Entries in HEPdata point to existing rivet analyses.

First Rivet runs

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_`
- List CMS analyses:
  `rivet --list-analyses CMS_`
- 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 an `.info` file which accompanies the analysis.

Running a simple analysis (standalone)

To avoid running a generator, we feed (gzipped) hepmc files to Rivet directly.

```
gunzip -c qcd.hepmc.gz | rivet -a MC_JETS
```

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

```
gunzip -c qcd.hepmc.gz | rivet -a MC_JETS -a MC_GENERIC
```

Running a simple analysis (standalone)

To avoid running a generator, we feed (gzipped) hepmc files to Rivet directly.

```
gunzip -c qcd.hepmc.gz | rivet -a MC_JETS
```

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

```
gunzip -c qcd.hepmc.gz | rivet -a MC_JETS -a MC_GENERIC
```

SHERPA, Herwig7, ATLAS and CMS software can run Rivet through memory transfer, no file I/O, faster.

Running a simple analysis (standalone)

To avoid running a generator, we feed (gzipped) hepmc files to Rivet directly.

```
gunzip -c qcd.hepmc.gz | rivet -a MC_JETS
```

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

```
gunzip -c qcd.hepmc.gz | rivet -a MC_JETS -a MC_GENERIC
```

SHERPA, Herwig7, ATLAS and CMS software can run Rivet through memory transfer, no file I/O, faster.

FYI this is how you run e.g. Pythia8 events through Rivet:
Sacrifice frontend to Pythia8
Filesystem pipe: `mkfifo fifo.hepmc`
NB. A FIFO/pipe has to live in a non-AFS directory.

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

Plotting histograms

ROOT didn't meet our needs/aspirations :-(

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

Rivet 2 uses our (nice!) system called YODA – **http://yoda.hepforge.org**

YODA data format is plain text and stores all second-order statistical moments: can do full stat merging, including details like weighted focus inside bins. General annotation system for metadata – styling, notes, whatever.

Command line 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/gv/xpdf…

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

Running a data analysis

For example, the ATLAS 13 TeV Minimum Bias analysis:

```
rivet --show-analysis ATLAS_2016_I1419652
```

Or, the CMS 13 TeV charged hadron analysis:

```
rivet --show-analysis CMS_2015_I1384119
```

Note: tab completion for `rivet` options and analysis names — not in docker run though.

```
gunzip -c qcd.hepmc.gz | rivet -a CMS_2015_I1384119 -a
ATLAS_2016_I1419652
```

And plot, much as before:

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

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

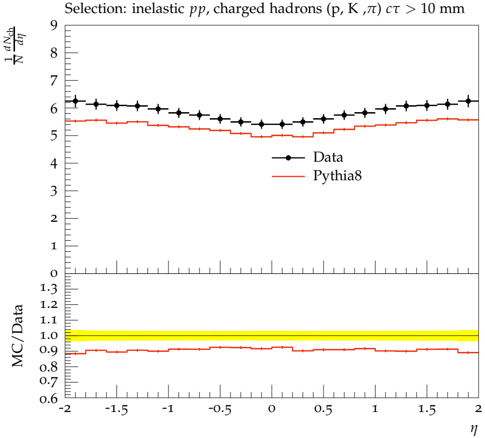
Example output

rivet-mkhtml Rivet.yoda:Pythia8

```
BEGIN YODA_HISTO1D /CMS_2015_I1384119/d01-x01-y01
IsRef=1
Path=/CMS_2015_I1384119/d01-x01-y01
ScaledBy=1.00000000000000005e-04
Title=
Type=Histo1D
XLabel=
YLabel=
# Mean: 2.047907e-03
# Area: 2.128100e+01
# ID         ID          sumw          sumw2          sumwx          sumwx2          numEntries
Total          Total          2.128100e+01          2.128100e-03          4.358152e-02          2.926641e+01          212810
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
# xlow          xhigh          sumw          sumw2          sumwx          sumwx2          numEntries
-2.000000e+00          -1.800000e+00          1.105200e+00          1.105200e-04          -2.100061e+00          3.994160e+00          11052
-1.800000e+00          -1.600000e+00          1.111800e+00          1.111800e-04          -1.889834e+00          3.216089e+00          11118
-1.600000e+00          -1.400000e+00          1.090400e+00          1.090400e-04          -1.636227e+00          2.458917e+00          10904
-1.400000e+00          -1.200000e+00          1.100500e+00          1.100500e-04          -1.430107e+00          1.862041e+00          11005
-1.200000e+00          -1.000000e+00          1.074200e+00          1.074200e-04          -1.181831e+00          1.303798e+00          10742
-1.000000e+00          -8.000000e-01          1.063700e+00          1.063700e-04          -9.578032e-01          8.659895e-01          10637
-8.000000e-01          -6.000000e-01          1.048100e+00          1.048100e-04          -7.341084e-01          5.176510e-01          10481
-6.000000e-01          -4.000000e-01          1.037100e+00          1.037100e-04          -5.192426e-01          2.634387e-01          10371
-4.000000e-01          -2.000000e-01          1.015100e+00          1.015100e-04          -3.042990e-01          9.460527e-02          10151
-2.000000e-01          0.000000e+00          9.916000e-01          9.916000e-05          -9.947841e-02          1.330623e-02          9916
                                        .
                                        .
                                        .
 # END YODA_HISTO1D
```

Example output

`rivet-mkhtml Rivet.yoda:Pythia8`



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 – details handled in the library + expressive API functions.

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

Mostly "normal":

- Typical init/exec/fin structure
- Histogram booking normal here, but no titles, labels, etc.: use `.plot` file
- Rivet's own Particle, Jet and FourMomentum classes: some nice things like `abseta()` and `abspid()`, decay chain searching, and auto-conversion to/from fastjet::PseudoJet
- Use of *projections* for computations, with a bit of magic – this is where the caching happens
- Projections are *declared* with a string name, and later are *applied* using the same name
- Final state projections are central: compute from final state or physical decayed particles

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**

Particle finders & final-state projections

Rivet is mildly obsessive about only calculating things from final state objects. Accordingly, a *very* important set of projections is those used to extract final state particles: these all 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.
- `VetoedFinalState` finds particles *other* than specified.
- `VisibleFinalState` excludes invisible particles like neutrinos, LSP, etc.

Most FSPs can take another FSP as a constructor argument and augment it. In the near future FSPs should be able to take arbitrary combinations of kinematic cuts as a single argument.

Using an FSP to get all final state particles

```
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

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 coming…

Jet tagging

Previously used a very inclusive tagging definition based on hadron parentage:

- `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 define the truth tag:

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

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.

Jets (1)

There are many more projections, but one more important set which we'd like to dwell on is those to construct jets. `JetAlg` is the main projection interface for doing this, but almost all jets are actually constructed with FastJet, via the explicit `FastJets` projection.

The `FastJets` constructor defines the input particles (via a `FinalState`), as well as the jet algorithm and its parameters:

```
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");
```

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

Jets (2)

Then get the jets from the jet projection, 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!
  }
}
```

Check out the `Rivet/Math/MathUtils.hh` header for more handy functions like `deltaR`.

Jets (3)

For substructure analysis Rivet doesn't provide extra 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);
  ...
}
```

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 try jets, leptons.

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.

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)`**.

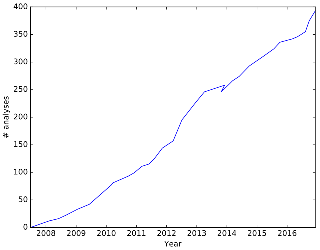
NB. Extra bool argument for using ref data x vals for **`Scatter2D`**s

Rivet + fast-sim for BSM searches

BSM analysis coverage

Currently $\sim$ 427 analyses total & $\sim$ 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 $\rightarrow$ 2.x migration.
Note recent acceleration!

- $\Rightarrow$ 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!**

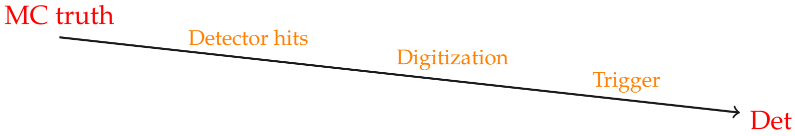
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies

MC truth

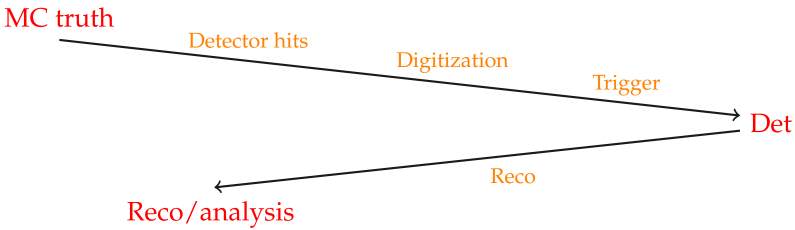
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



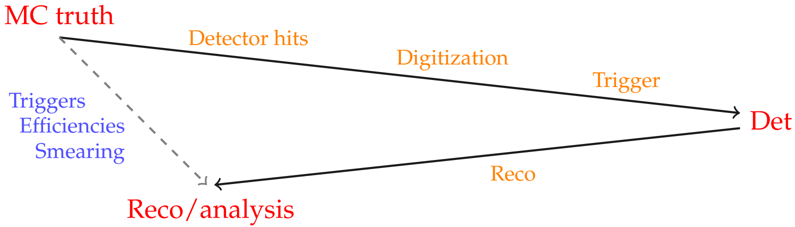
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



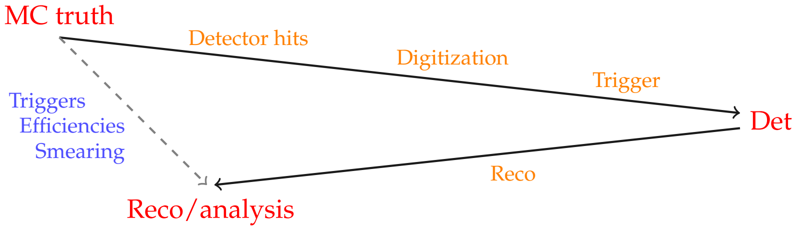
BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies



BSM & detector effects

Explicit fast detector simulation vs. smearing/efficiencies

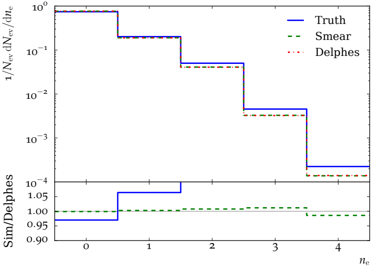


- **Explicit fast-sim takes the "long way round".**
- **Reco already reverses most detector effects!**
- Reco calibration to MC truth: smearing is a few-percent effect
- (Lepton) efficiency & mis-ID functions dominate – and are tabulated in both approaches
- Smearing is more flexible: effs change with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

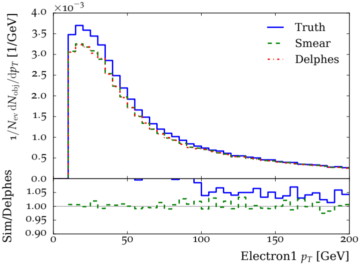
Smearing vs. fast sim vs. MC truth

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

Electron multiplicity



Leading electron p_T

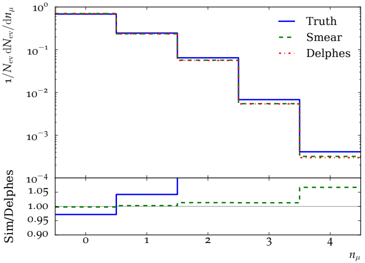


Note major lepton shifts from blue truth to green smeared: difference w.r.t red DELPHES very small

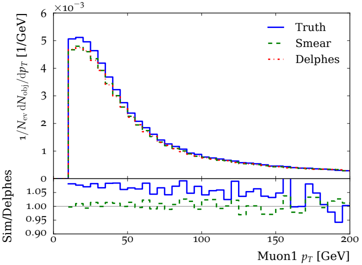
Smearing vs. fast sim vs. MC truth

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

BSM & detector effects (II) ⇒ Rivet 2.5

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 2.5 fast-sim

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(es1, ELECTRON_EFF_ATLAS_RUN2, ELECTRON_SMEAR_ATLAS_RUN2);
declare(recoes, "Electrons");

FastJets js1(FastJets::ANTIKT, 0.6, JetAlg::DECAY_MUONS);
SmearedJets js2(js1, 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 2.5 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"`

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 for search analyses

Search analyses typically do a lot more "object filtering" than measurements. Rivet 2.5 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"`

Examples:

```
// 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 for search analyses

Search analyses typically do a lot more "object filtering" than measurements. Rivet 2.5 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"`

Examples:

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

Selection tools for search analyses

Search analyses typically do a lot more "object filtering" than measurements. Rivet 2.5 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"`**

Examples:

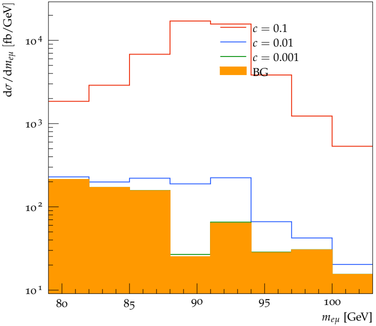
```
// Discard muons close to remaining jets
const Particles isomus = filter_discard(mus, [&](const Particle& m) {
    for (const Jet& j : isojets) {
      if (deltaR(j,m) > 0.4) continue;
      if (j.particles(Cuts::abscharge > 0 && Cuts::pT > 0.4*GeV).size() > 3)
        return true;
    }
    return false;
  });
```

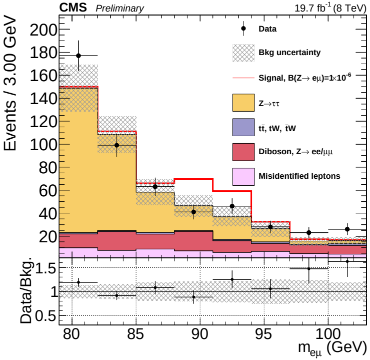
That’s all, folks

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 ATLAS analysis bugs have been found via Rivet!
- Strongly encouraged/required by ATLAS (and CMS?) physics groups. Integrated with ATLAS and CMS 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!**

- Signal and background files for $Z \to e\mu$
- Write analysis code to reconstruct $m_{e\mu}$, e.g. **`CMS PAS EXO-13-005`**
- Example code from downloads page, **`MC_ZMUE.cc`**





- You can use **`yodamerge -stack`** to add signal and background histos.

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 Andy 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.