Analysis prototyping, preservation and recasting with Rivet

Christian Gütschow
(stealing most material from Andy Buckley)

Rivet tutorial, DESY

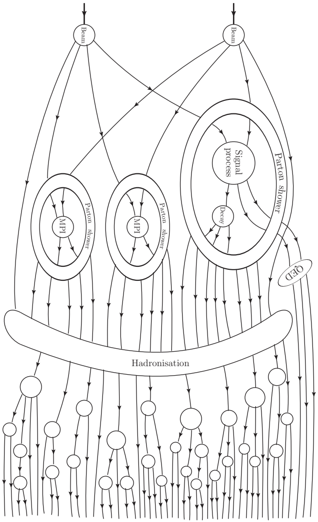
25 October 2017

Introduction

- **R**obust **I**ndependent **V**alidation of **E**xperiment and **T**heory
  - generator-agnostic, efficient and fast
- quick, easy and powerful way to get physics plots from lots of MC generators
  - only requirement: use HepMC event record
- lightweight way to exchanging analysis details and ideas
- Rivet has become the LHC standard for archiving LHC data analyses
  - focus on unfolded measurements more than searches,
    but **fast detector simulation now also intrinsic to Rivet**
  - key input to MC validation and tuning – increasingly comprehensive coverage
  - also “recasting” of SM and BSM data results on to new/more general new-physics models

Design philosophy

- Rivet operates on HepMC events, intentionally unaware of who made them
  - event graph looks very different depending on the generator
  - reconstruct resonances, dress leptons, avoid partons
  - makes you think about physics & helps find analysis bugs/ambiguities
- C++ library with Python interface & scripts
- write your analysis **plugin** without needing to rebuild Rivet
- comes with plenty of tools to make work flow easy
- computation caching for efficiency
- histogram syncing: keep code clean and clear



Rivet setup

- latest version is 2.5.4
  - requires C++11
- local installation using the **bootstrap script**
  - `wget http://rivet.hepforge.org/hg/bootstrap/raw-file/2.5.4/rivet-bootstrap`
  - `bash rivet-bootstrap`
- docker container: `docker pull hepstore/rivet:2.5.4`
- can also pick up latest version from Genser/LCG build area
  - `source /afs/cern.ch/sw/lcg/releases/LCG_87/Python/2.7.10/x86_64-slc6-gcc49-opt/Python-env.sh`
  - `source /afs/cern.ch/sw/lcg/releases/LCG_87/MCGenerators/rivet/2.5.4/x86_64-slc6-gcc49-opt/rivetenv.sh`

Viewing available analyses

- `rivet` command line tool to query available analyses
- Rivet knows all sorts of details about its analyses
  - list available analyses: `rivet --list-analyses`
  - list available ATLAS analyses: `rivet --list-analyses ATLAS_`
  - show some pure-MC analysis' full details: `rivet --show-analysis MC_ZJETS`
- PDF and HTML documentation is also built from this info, so is always synchronised
- analysis metadata is provided via the analysis API and usually read from an `.info` file which accompanies the analysis

Running Rivet

- Rivet 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
  - `rivet -a MC_JETS input.hepmc`
  - unfinalised histos are written every 1000 events (can monitor progress through the run)
  - killing with `Ctrl-C` is safe (finalizing is run)
- 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

Example output

```
# 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 needs/aspirations
  - bin width issues, bin gaps unhandled, object ownership nightmare, thread-unsafety
- Rivet uses alternative 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`
- then view with a web browser/file browser/evince/...
- a `--help` option is available for all Rivet scripts

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
  - 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

Writing an analysis

- writing an analysis generally more involved, but C++ interface 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
  `https://rivet.hepforge.org/code/dev/MC__ZINC_8cc_source.html`
- code is "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 iswhere 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

- projections 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()
  ...
}
```

- best to get a handle to the applied projection as a `const` reference to avoid unnecessary copying
- 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`
  - `FinalState` finds all final state particles in a given range, with a given pT 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

Using an FSP to get all final state particles

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

➜ more complex projections like `DressedLeptons`, `FastJets`, `ZFinder`, `TauFinder`, . . . implement experimental 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 I

- `JetAlg` is the main projection interface to construct jets, but almost all jets are actually constructed with `FastJet`, via the explicit `FastJets` projection
- `FastJets` constructor defines the input particles (via a `FinalState`), as well as the jet algorithm and its parameters:

```
const FinalState fs(Cuts::abseta < 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 II

- 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 III

- 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);
  ...
}
```

Jet tagging

- previously used 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)`

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**
- histograms have the usual `fill(value, weight)` method for use in `analyze()` method
- there are `scale()`, `normalize()` and `integrate()` methods for use in `finalize()`
- fill weight is important (!) for kinematic enhancements, systematics, counter-events, etc.
- use `evt.weight()` (until automatic multiweight support ...)

Histogram autobooking

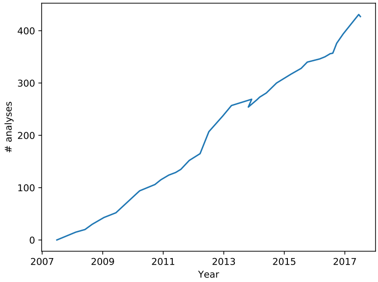
- histogram autobooking is a means for getting your Rivet histograms binned with the same bin edges as used in the experimental data that you'll be comparing to
- to use autobooking, just call the booking helper function with only the histogram name (check that this matches the name in the reference `.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 the 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)`

Writing, building & running your own analysis

- prepared some $Z \to \mu\mu$ events in `~cgutscho/public/rivetDESY/material.tar.gz` (on both lxplus or NAF)
- unpack, set up Rivet using script if necessary: `./setupRivet2.5.4`
- 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
- implement dimuon selection using e.g. `ZFinder` projection
- when done, you can either compile directly with `g++`, using `rivet-config` script as a compile flag helper, or run `rivet-buildplugin RivetMY_TEST_ANALYSIS.so 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)
- Let's see some Z resonances!

Analysis preservation in Rivet

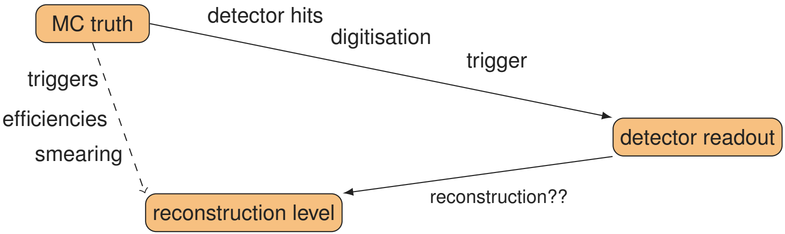
- currently ∼430 analyses total (∼ 230 LHC analyses alone)
- until recently only 27 dedicated BSM searches and BSM-sensitive SM measurements
- SM focus on unfolded observables, not sufficient for most BSM studies



- Rivet 2.5.0 introduced detector smearing machinery
- added many real-world examples of how to write BSM routines
- also added tools to help with object filtering, cutflows, etc.
- **Rivet is in good shape for preserving new physics searches!**

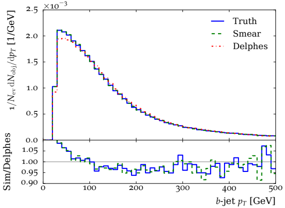
BSM & detector effects

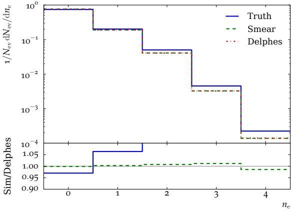
- explicit fast detector simulation vs. smearing/efficiencies

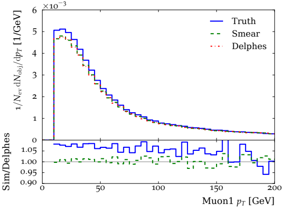


- explicit fast-sim takes the “long way round”
- reconstruction already reverses most detector effects!
- reco calibration to MC truth: smearing is a few-percent effect
- (lepton) efficiency & mis-ID functions dominate – and are tabulated in both approaches
- smearing is more flexible: efficiencies change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

Smearing vs fast sim vs MC truth







- ➜ flexibility of detector simulation 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, . . .
- ➜ Rivet detector simulation as efficiencies + smearing localised per analysis
  - ➜ available since version 2.5.0

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"`

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

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

...

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

- small tweak planned, to unify eff/mod functions and give user control of operator ordering

Selection tools for search analyses

- searches 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"`

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));
});

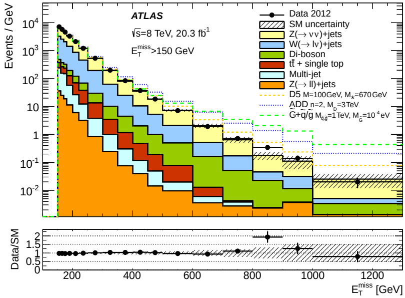
// Discard muons close to remaining jets
const Particles isomus = filter_discard(mus, [&](const Particle& m) {
  for (const Jet& j : isojets) {
    if (deltaR(j,m) > 0.4) continue;
    if (j.particles(Cuts::abscharge > 0 && Cuts::pT > 0.4*GeV).size() > 3)  return true;
  }
  return false;
});
```

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 e.g. ATLAS physics groups
- 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!

Final exercise: reinterpretation

- ATLAS 8 TeV monojet search provides measured data and SM background estimates
- `https://hepdata.net/record/ins1343107`
- `https:/arxiv.org/abs/1502.01518`



- you can find both data (`ATLAS_2015_I1343107.yoda`) and background (`BG.yoda`) in `Monojet` directory
- try to implement the monojet selection (Table 2) and run over one of the Dark Matter models in `Events`
- can use provided script to combine signal and background:
  `python addSignalAndBackground.py S.yoda SplusB.yoda`