Analysis prototyping, preservation, reinterpretation, tuning, . . . , with the Rivet toolkit

MPI@LHC, Perugia, 14 December 2018
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**
  - 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 code to complement the data analysis is increasingly expected of experiment 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 code to complement the data analysis is increasingly expected of experiment 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 + *lots* of analyses
≈ 430 built-in! ≈ 50 are pure MC, and some double-counting

▶ Easy and powerful way to get physics numbers & plots from *any* MC gen
Rivet

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

~ 430 built-in! ~ 50 are pure MC, and some double-counting

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

▶ LHC standard for preserving data analyses: standard in ATLAS & CMS SM
Rivet

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

∼ 430 built-in! ∼ 50 are pure MC, and some double-counting

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

▶ LHC standard for preserving data analyses: standard in ATLAS & CMS SM

▶ Origins in SM, and particularly QCD for MCs – extended for search preservation since v2.5 by adding detector transfer-function features
Rivet is an analysis system for MC events + *lots* of analyses

~ 430 built-in! ~ 50 are pure MC, and some double-counting

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

- LHC standard for preserving data analyses: standard in ATLAS & CMS SM

- Origins in SM, and particularly QCD for MCs – extended for search preservation since v2.5 by adding detector transfer-function features

- C++ library with Python interface, analyses are plugins, code is “clean”
Rivet

Rivet is an analysis system for MC events + lots of analyses
\(~ 430 \text{ built-in}! \sim 50\) are pure MC, and some double-counting

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

- LHC standard for preserving data analyses: standard in ATLAS & CMS SM

- Origins in SM, and particularly QCD for MCs – extended for search preservation since v2.5 by adding detector transfer-function features

- C++ library with Python interface, analyses are plugins, code is “clean”

- “If you can’t write a Rivet analysis for it, it’s probably unphysical”!

Andy Buckley
Generator independence

A Pythia8 $t\bar{t}$ event visualised from HepMC output:

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

Lots of analyses, but we’re still missing a lot! You can help…

NEW! Semi-automatic Rivet LHC analysis wishlist

Rivet LHC analysis coverage (no searches)
Rivet analyses exist for 214/1323 papers = 16%. 144 priority analyses required.
Total number of Inspire papers scanned = 1916, at 2018-12-12
Breakdown by identified experiment (in development):

<table>
<thead>
<tr>
<th>Key</th>
<th>ALICE</th>
<th>ATLAS</th>
<th>CMS</th>
<th>LHCb</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rivet wanted</td>
<td>294</td>
<td>135</td>
<td>276</td>
<td>140</td>
<td>454</td>
</tr>
<tr>
<td>Rivet REALLY wanted</td>
<td>39</td>
<td>36</td>
<td>59</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Rivet provided</td>
<td>11/213 = 5%</td>
<td>119/254 = 47%</td>
<td>69/243 = 27%</td>
<td>11/153 = 7%</td>
<td>8/462 = 2%</td>
</tr>
</tbody>
</table>

---

Andy Buckley
### Rivet setup

#### Docker

VM-like pre-prepared environments: avoid platform issues, integrates well with host. Instructions at [https://rivet.hepforge.org/trac/wiki/Docker](https://rivet.hepforge.org/trac/wiki/Docker)

```bash
docker pull hepstore/rivet-tutorial
docker run -it -v $PWD:/out hepstore/rivet-tutorial
```

#### Local install

Easy to install using our *bootstrap script*:

```bash
bash rivet-bootstrap
```

Needs valid compiler (C++11), etc. environment

#### Run from LCG

```bash
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
**Command-line interface**

`rivet` and other command line tools to query and run routines

- List available analyses:
  ```shell
  rivet --list-analyses
  ```

- List ATLAS analyses:
  ```shell
  rivet --list-analyses "ATLAS|CMS"
  ```

- Show some pure-MC analyses’ full details:
  ```shell
  rivet --show-analysis MC_
  ```

Same metadata and API docs online at [http://rivet.hepforge.org](http://rivet.hepforge.org)

All Rivet commands start with `rivet-`, so tab-complete lists them all
Running existing analyses

To avoid huge files, we get the events from generator to Rivet by writing HepMC (from Py8) to a filesystem pipe

```
$ mkfifo fifo.hepmc
$ run-pythia -n 200000 -e 8000 -c Top:all=on -o fifo.hepmc &
$ rivet fifo.hepmc -a MC_TTBAR,MC_JETS,MC_GENERIC
             -a ATLAS_2015_I1404878,CMS_2016_I1473674
$ rivet-mkhtml Rivet.yoda:’Pythia 8 $t\bar{t}$’
```

By default *unfinalised* histos are written every 1000 events: monitor progress through the run. Killing with Ctrl-c is safe: finalizing is run...
“YODA” stats library — http://yoda.hepforge.org
Bin-width handling, bin gaps, object ownership, thread-safety ⇒ non-ROOT histogramming

▶ Separation of stats from presentation: plotting via make-plots script
▶ Text-based data format with all second-order stat moments: full stat merging up to all means and variances
▶ YAML metadata and zipped read/write from v1.7.0
▶ Being gradually extended to handle more complex physics data types

CLI tools: yodals, yodadiff, yodamerge, yodascale, yoda2root, etc.
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 [https://rivet.hepforge.org/analyses/MC_GENERIC.html](https://rivet.hepforge.org/analyses/MC_GENERIC.html)
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 https://rivet.hepforge.org/analyses/MC_GENERIC.html

Code is “mostly normal”:

- Typical init/exec/finalize loop structure
- Histograms ~normal; titles, etc. → external .plot file
- Particle, Jet and FourMomentum classes with some nice things like \texttt{abseta()} and \texttt{abspid()}, constituents, decay-chain searching, and compatibility with FastJet objects
- Use of \textit{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:

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

Application in the analyze method via the same name:

```cpp
void analyze(const Event& evt) {
    ...
    const SomeProjBase& mysp = apply<SomeProj>(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 $\eta$ 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

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

```cpp
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.
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
One more important projection set is those which find \textit{jets}

There's a \texttt{JetAlg} abstract interface, but almost always use FastJet, via \texttt{FastJets}

Define the input particles (via a \texttt{FinalState}), and the jet alg & params:

```cpp
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:

```cpp
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 \texttt{Rivet/Math/MathUtils.hh}!
Jet flavour

FastJets automatically ghost-tags jets using $b$ and $c$ hadrons (and $\tau$'s):

- if (myjet.bTagged()) ...
- if (myjet.bTags().size() > 1) ...

And you can use cuts to refine the truth tag:

- myjet.bTagged(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

```cpp
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`:

```cpp
d23 = cs.exclusive_subdmerge(jetproj.jetsByPt[0], 2)
```
Let’s start with a simple “particle analysis”, just plotting some simple particle properties like $\eta$, $p_T$, $\phi$, 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
Detector effects

Normal in SM, top, etc. measurements to unfold detector effects. Usually “uneconomic” to do that for BSM searches

Explicit fast detector simulation vs. smearing/efficiencies:

- (Private) reco algorithms already reverse most detector effects
- Reco calibration to MC truth, so kinematics usually subleading
- Efficiency & mis-ID effs dominate – tabulated in all fast-sims
- ⇒ flexible parametrisation: effs change with analysis phase-space, experiment reco-code version, collider run, …
  and need to guarantee stability for preservation
Using Rivet’s fast-sim tools

Smearing is provided as “wrapper projections” on normal particle, jet, and MET finders.

Smearing configuration via efficiency/modifier functions.

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

Examples:

```cpp
FinalState es1(Cuts::abseta < 5 && Cuts::abspid == PID::ELECTRON);
SmearedParticles es2(es1, ELECTRON_EFF_ATLAS_RUN2, ELECTRON_SMEAR_ATLAS_RUN2);
declare(es2, "Electrons");

FastJets js1(FastJets::ANTIKT, 0.6, JetAlg::DECAY_MUONS);
SmearedJets js2(fj, JET_SMEAR_ATLAS_RUN2, JET_EFF_BTAG_ATLAS_RUN2);
declare(js2, "Jets");

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

Standard global functions here, but private fns or inline lambdas better when possible
Selection tools for search analyses

Search analyses typically do a lot more “object filtering” than measurements. Lots of tools to express complex logic neatly:

- **Filtering functions**: `filter_select(const Particles/Jets&, FN), filter_discard(...) + ifilter_* in-place variants`

- **Functors** for common “stateful” filtering criteria:
  `PtGtr(10*GeV), EtaLess(5), AbsEtaGtr(2.5), DeltaRGtr(mom, 0.4), ParticleEffFilter(FN), ...
  
  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"`
BSM hands-on

Look at the source code in `TESTDET.cc`: does it make sense?

- **Build & run like:**
  
  ```
  $ rivet-buildplugin TESTDET.cc
  $ run-pythia -n 200000 -e 13000 -o fifo.hepmc -c SUSY:all=on
  -c SLHA:file=gg_g1500_chi100_g-ttchi.slha &
  $ rivet --pwd -a TESTDET -H bsm.yoda fifo.hepmc -lAnalysis=DEBUG
  ```

- **Split and compare the particle- and reco-level observables:**
  
  ```
  $ bash truerecosplit.sh bsm.yoda
  $ rivet-mkhtml bsm-*.yoda -m '/TESTDET'
  ```

- **Try adding a constant 70\% b-tag efficiency to the jets:**
  
  ```
  JET_BTAG_EFFS(0.7) OR
  (const Jet& j) return j.bTagged() ? 0.7 : 0.0; .
  ```

- **Try the same with `CMS_2017_I1594909.cc`; browse the file with `yodals -v` to see the the CMS signal-region counts for recasting**

Andy Buckley
Contur: BSM limit-setting using Rivet SM analyses

Contur is a layer on top of Rivet to do statistical interpretation of injected BSM signal to “Standard Model” phase spaces.

▶ **Idea:** make use of the full set of Rivet analyses to constrain new physics models. Modelling inclusivity also important: a strength of Herwig 7

▶ **Benefits:** model-agnostic and very quick. Can study many possible signatures at the same time

▶ **Current constraints (in progress):** SM MC is complex ⇒ assume data = SM 
Single-bin limits within manual analysis groupings in lieu of full correlations. 
Working to include SM predictions and uncertainties
That’s all, folks
Rivet is a user-friendly MC analysis system for prototyping and preserving data analyses. It allows theorists to use analyses for model development & testing, MC tuning, and BSM recasting. Also a very useful cross-check: quite a few analysis bugs have been found via Rivet. Rivet supports detector simulation for BSM search preservation. Contributions and team membership are all very welcome. Twice-annual Rivet hackathons in nice places! Funded 3+ month MCnet studentships are available.

Rivet is a great way to get a feel for MC physics, prototype analyses, and work on SM & BSM phenomenology studies with theorists.