

Implementing a CMS SUSY search with a Single Lepton, Missing Energy and b-jets in Rivet

Andrew Peter McMahon, University of Glasgow, UK

September 5, 2012

Abstract

We present the inclusion of a CMS SUSY search analysis for final states containing a single lepton, missing energy and b-jets in the Monte-Carlo validation tool Rivet. We discuss the philosophy and implementation of the Rivet package and the new analysis code written by the author. We then compare the results found using this code with results from previous CMS studies and discuss the level of agreement. Possible directions of future work with Rivet are also discussed.

Contents

1. Introduction	3
2. Supersymmetry	3
2.1. The Experimental Status of Supersymmetry	5
3. CMS	6
4. Implementing an Analysis in Rivet	7
5. Event Generation	7
5.1. Pythia	7
6. Results	7
6.1. H_T	7
6.2. Y_{MET}	8
6.3. Jet E_T	9
6.4. Isolated Jet Multiplicity and Isolated b-Jet Multiplicity	9
6.5. Y_{MET} vs H_T	10
7. Conclusions	11
A. Appendices	12
B. The basic structure of a Rivet code	12
B.1. Main Structure	13
B.2. Toy Analysis	14
C. Understanding Toy Analysis - Initialize	15
D. Understanding Toy Analysis - Analyze	15
E. Understanding Toy Analysis - Finalize and Completing the Code	17
F. Conclusions	18

1. Introduction

The wealth of experimental data coming from the LHC means that it is becoming increasingly important to have robust and accurate methods for simulating high energy collisions. In order for event generators to be accepted as valid tools in this endeavour it is imperative to have their methods tuned and updated in response to previously gathered data and this is precisely what the Rivet [1] package has been designed to facilitate. Rivet is an extensive toolkit written in C++ which allows us to implement direct comparisons between simulated Monte-Carlo generator data and experimental data (and indeed it also allows comparison between Monte-Carlo generators themselves). It does this in a generator independent way by only requiring that the format for event files it reads be HepMC [3] (which most commonly used event generators can accomodate), all of its subsequent features being completely independent of where the data came from. Therefore Rivet is a powerful addition to the growing list of sophisticated software at use in the High Energy Physics community.

In this paper we discuss the implementation of a new CMS SUSY analysis [2] in Rivet and compare the results found with studies done by the DESY CMS SUSY group. We show through a detailed implementation that we can obtain good agreement with previous generation runs performed by the CMS group here at DESY for use in [2].

2. Supersymmetry

The recent success of the LHC collaborations in discovering a new scalar boson resonance at ≈ 125 GeV has understandably dominated the current agenda in the particle physics community. Although this discovery is a major achievement for the LHC and indeed provides the required justification for construction of the largest and most complex machine ever built, it must be remembered that we expect the LHC not only to help elucidate the mechanism of Electroweak Symmetry Breaking, but also to provide insights on some of the conceptual problems of the Standard Model. We are searching for glimpses to the answers of questions such as: Why is the magnitude of the gauge Hierarchy between the ElectroWeak and Planck scales so huge? Do the couplings of the forces described by the Standard Model unite at some as of yet unknown energy scale? What is the particle content of Dark Matter? These questions are ones which theorists have attempted to answer by extending the Standard Model in various ways and of all of these extensions, none is more studied than Supersymmetry.

Supersymmetry is an attempt to reconcile the Hierarchy problem and the problem of unstable fluctuations in the calculation of the Higgs mass (due to top loops) in the Standard Model by hypothesising a mapping between bosonic and fermionic degrees of freedom. This equates to us assuming that for every fermionic particle (an electron for example) there is a corresponding boson (a selectron in this case) with all gauge quantum numbers identical except the magnitude of the intrinsic spin, which will differ

by a half. This very simple idea (compelling from an aesthetic point of view) provides a solution to almost all of the problems mentioned above. It does this in the following way:

1. The introduction of particles with identical quantum numbers but spins differing by a half allows dramatic cancellations to occur in the calculation of the Higgs boson mass in the Standard Model (due to the introduction of loops with a relative minus sign).
2. The introduction of these new degrees of freedom alters the running of the different couplings in the Standard Model, allowing unification at some high energy scale (the GUT scale).
3. Supersymmetric theories provide particle spectra in which the Lightest Supersymmetric Particle (LSP) is only weakly interacting and yet massive and stable, providing a Dark Matter candidate.

So these are the perks of introducing Supersymmetry into the Standard Model, but what are the downsides? Well for one there has been no observation of any Supersymmetric particles to date, so Supersymmetry must be a broken symmetry. Secondly, by introducing these new degrees of freedom we generate many new parameters in our theory (on the order of ≈ 100) so SUSY parameter space is very complicated. These two points taken together means that SUSY phenomenology should be very rich, but that the challenge of finding it is a very daunting one.

2.1. The Experimental Status of Supersymmetry

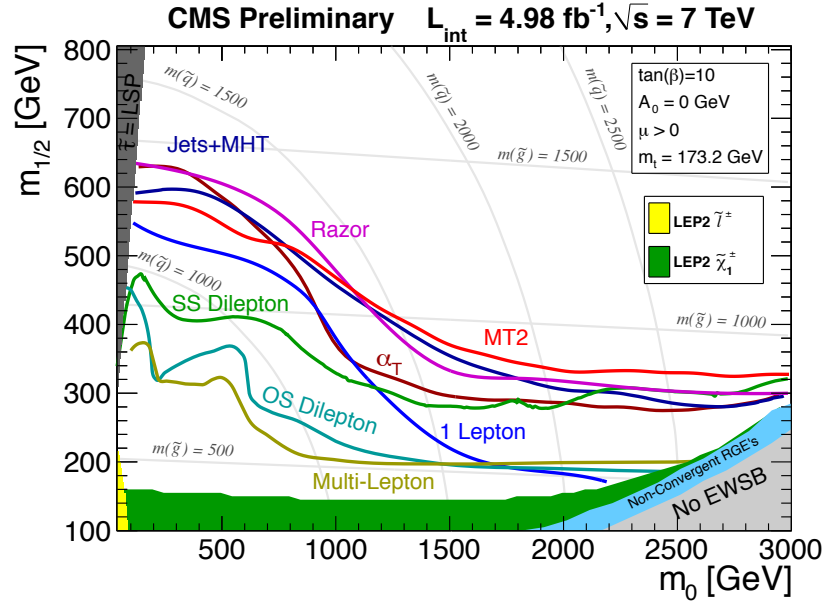


Figure 1: Observed limits from several 2011 CMS SUSY searches plotted in the CMSSM $(m_0, m_{1/2})$ plane.

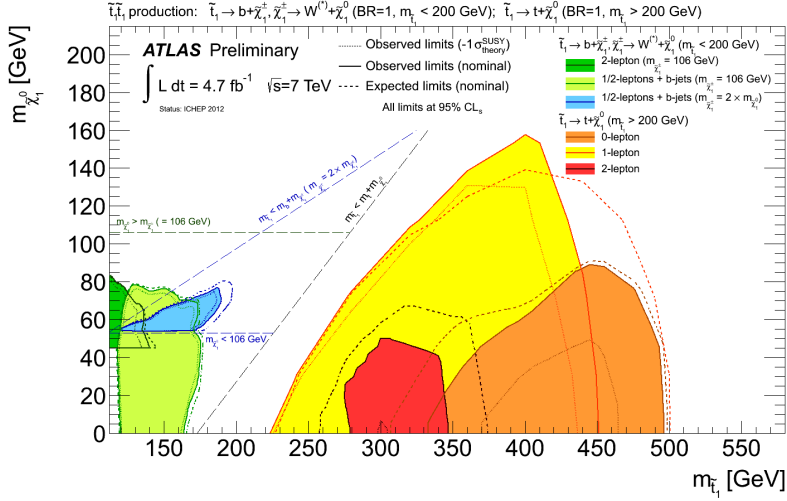


Figure 2: Summary of the five dedicated ATLAS searches for top squark (stop) pair production based on $4.7 fb^{-1}$ of pp collision data taken at $\sqrt{s} = 7$ TeV.

These plots show some recent exclusion limits placed on the Constrained Minimal Supersymmetric Standard Model by the CMS and ATLAS collaborations.

3. CMS

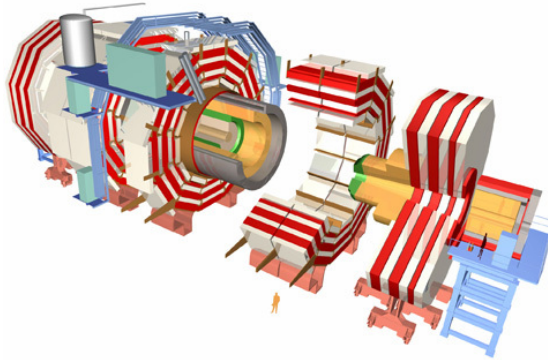


Figure 3: Piecewise view of the CMS detector.

The Compact Muon Solenoid is one of the two main multi-purpose detectors at the LHC and as such plays a hugely important role in the LHC physics programme. The centre of the detector contains a 13-m-long, 4 T superconducting solenoid with an inner diameter of 5.9m. The decision to include such a high magnetic field in the CMS detector was largely due to the desire for very good momentum resolution of charged particles (especially muons) and such a high field has meant that muon-chamber resolution and alignment requirements can be relatively relaxed (see pg. 8 of [5]). The detector consists of silicon tracking, electromagnetic and hadronic calorimeters (ECAL and HCAL

respectively) and the aforementioned muon-chambers. The detector is fully hermitic with a pseudorapidity coverage of $|\eta| < 5$.

4. Implementing an Analysis in Rivet

The main philosophy of Rivet is to act as an independent validation tool for Monte-Carlo event generators. In performing this task however it also acts as a very good platform for storage of different experimental analyses which have been performed on modern particle physics data. For a brief tutorial on how to implement an analysis in Rivet, the reader is referred to the appendices attached to this report.

5. Event Generation

In generating event samples for our current study we utilized the event generator Pythia 6.4 [4]. We generated 100,000 $t\bar{t}$ events with up to one extra parton in the final state.

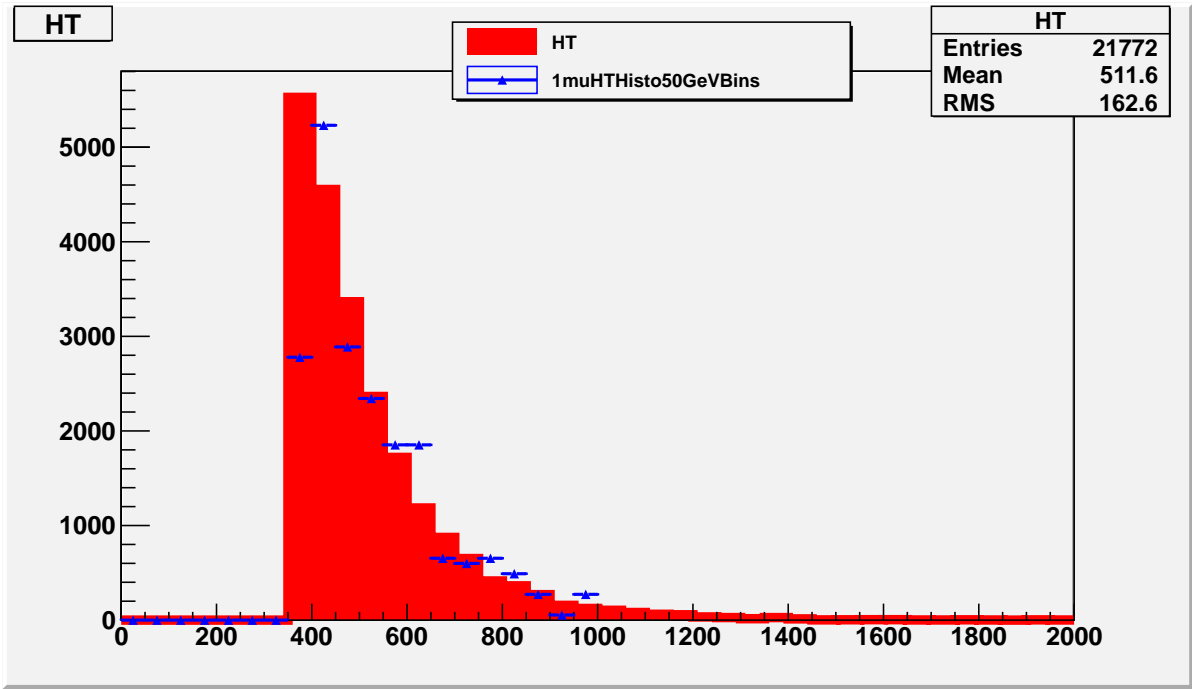
5.1. Pythia

Pythia [4] is a multi-purpose high energy physics Monte-Carlo event generator. Like all Monte-Carlo event generators it combines analytical results and phenomenological models with probabilistic methods in order to provide a good model of particle collisions at high energy. We will not go into detail about the methods employed in event generators nor much of the detailed physics of Pythia, which is beyond the scope of this report.

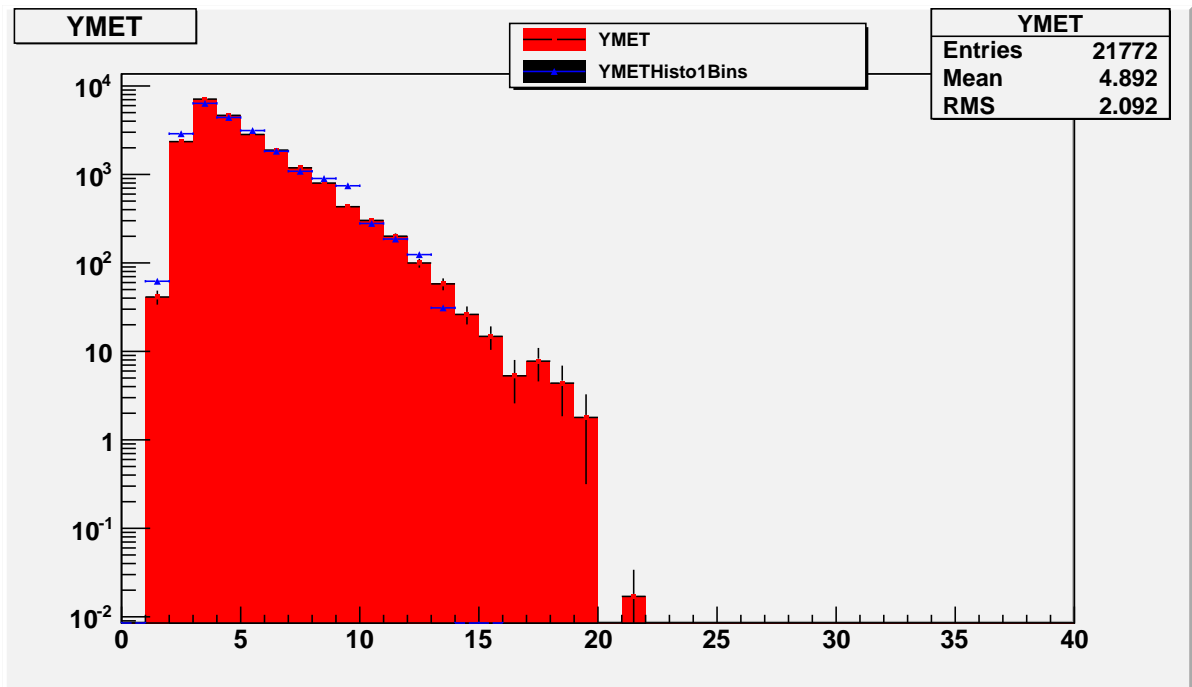
6. Results

In this section we shall compare the analyses of the Rivet code implemented by the author and that performed by the SUSY group here at DESY over the past year. We shall show plots of H_T , Y_{MET} , Jet E_T , Isolated Jet Multiplicity, Isolated b-Jet Multiplicity and finally Y_{MET} vs. H_T . Red histograms are results found by the SUSY group and blue points represent data found using the Rivet code. The histograms have been scaled in order to have the same area.

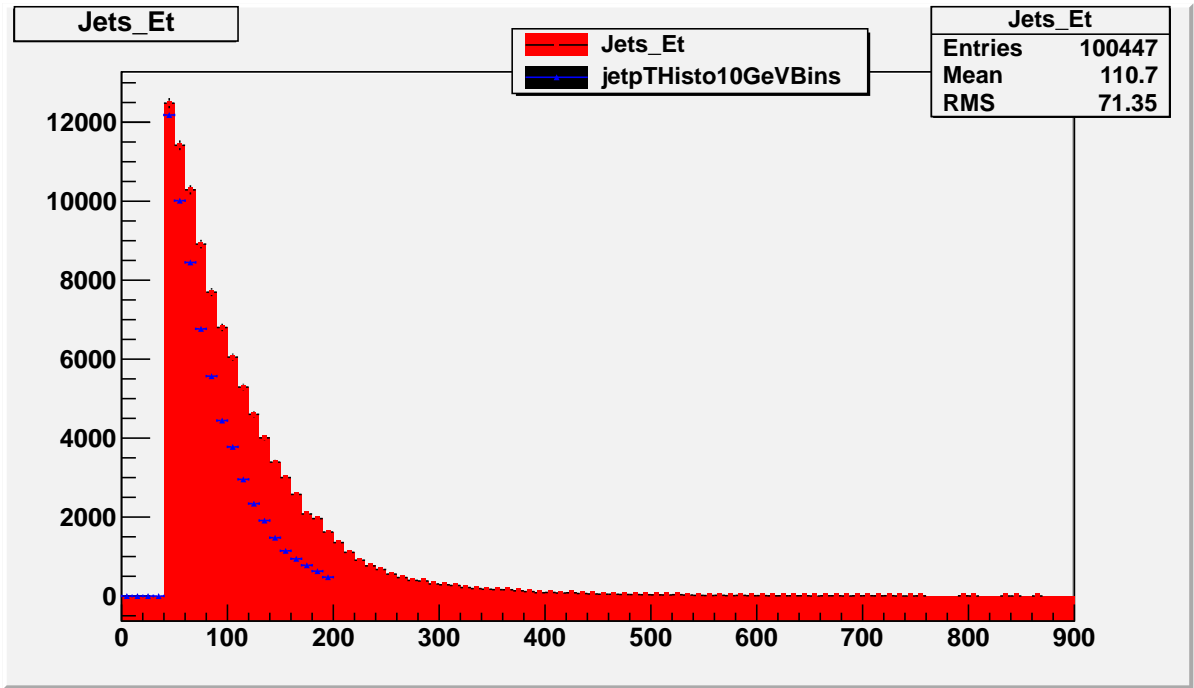
6.1. H_T



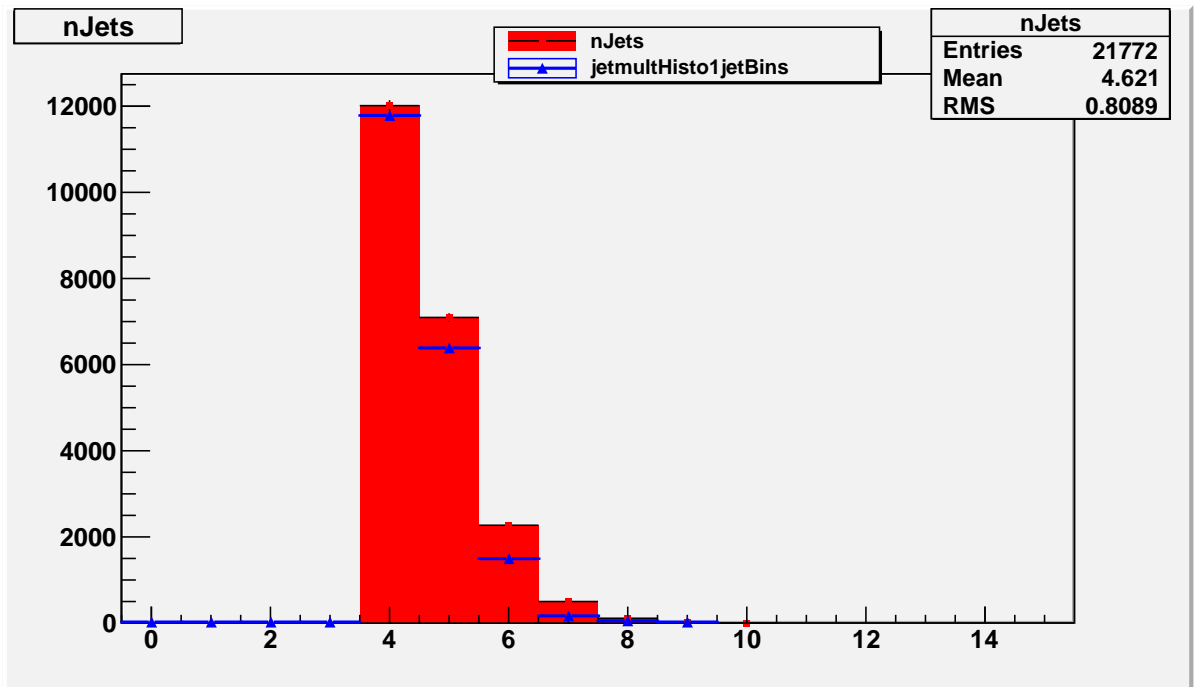
6.2. Y_{MET}

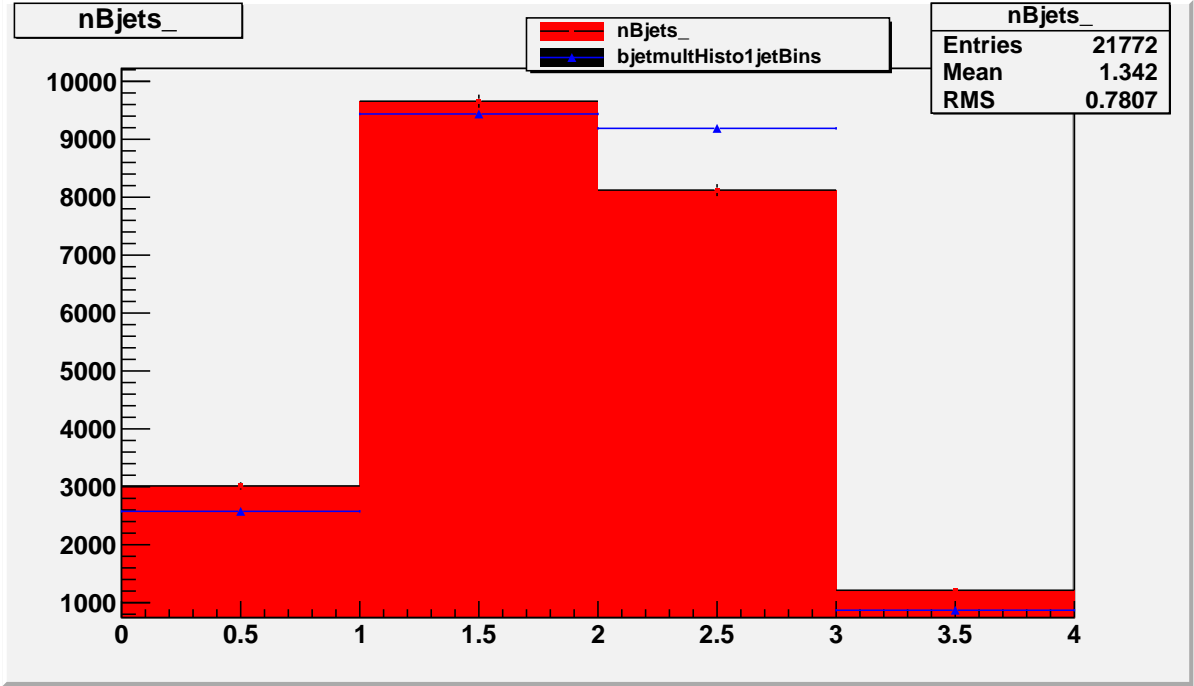


6.3. Jet E_T

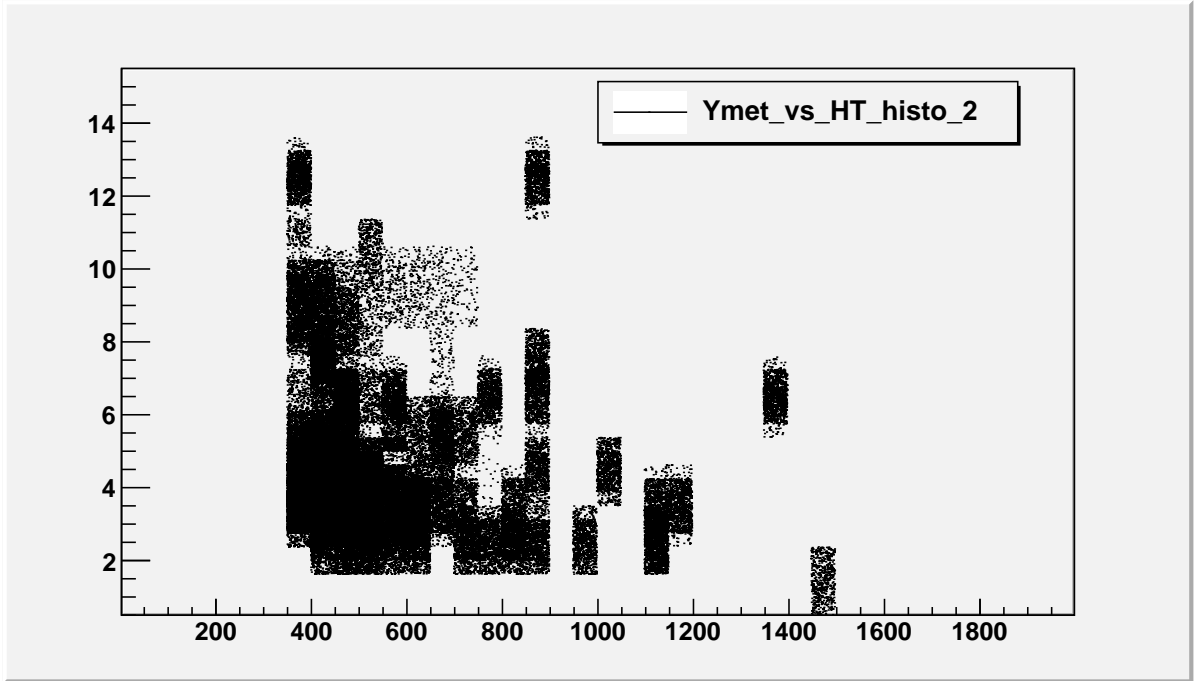


6.4. Isolated Jet Multiplicity and Isolated b-Jet Multiplicity





6.5. Y_{MET} vs H_T



In general these results show good agreement between the Rivet code and the dedicated physics analysis codes implemented by the CMS SUSY group here at DESY over the past year. We do however see some slight differences, for example in Jet E_T , which we believe could be down to the use of a different matrix element generator (the Author has

used Pythia straight out of the box whereas the SUSY group used Madgraph samples passed to Pythia for fragmentation). Essentially though, a rather simple Rivet code seems to have captured the main points of a dedicated physics analysis.

7. Conclusions

In this paper we have discussed the implementation of a CMS SUSY analysis in the software package Rivet. We have shown results found with the code and compared these to results used in the analysis in [2]. In general we see quite good agreement between these results, but the author believes that further improvement could be obtained by using the same matrix element and parton level information generator (i.e Madgraph [7]).

References

- [1] Rivet User Manual, arXiv:1003.0694v6 [hep-ph]. *Andy Buckley, Jonathan Butterworth, Leif Lönnblad, Hendrik Hoeth, James Monk, Holger Schulz, Jan Eike von Seggern, Frank Siegert, Jan Eike von Seggern.*
- [2] Search for Supersymmetry in Final States with a Single Lepton, B-Jets and Missing Transverse Energy in Proton-Proton Collisions at $\sqrt{s} = 7$ TeV. *The CMS Collaboration.*
- [3] Comput. Phys. Commun. 134 (2001) 41. *M. Dobbs and J.B. Hansen.*
- [4] Pythia 6.4 Physics and Manual, arXiv:hep-ph/0603175v2. *Torbjörn Sjöstrand, Stephen Mrenna, Peter Skands.*
- [5] http://cdsweb.cern.ch/record/922757/files/ptdr1_final_colour.pdf. *CMSCollaboration.* <https://twiki.cern.ch/twiki/bin/view/CMSPublic/PhysicsResults>
- [6] <https://twiki.cern.ch/twiki/bin/view/AtlasPublic>
- [7] <http://madgraph.hep.uiuc.edu/>

A. Appendices

B. The basic structure of a Rivet code

The basic features of a Rivet analysis code are very simple. On the next page show the main structure of a Rivet code with a large amount of detail omitted:

B.1. Main Structure

```
#include ‘ ‘...’ ’
#include ‘ ‘...’ ’

namespace Rivet {
    class Name_Of_Analysis : public Analysis {
    public:
        RA4B_Newest_Analysis() : Analysis(‘ ‘RA4B_Newest_Analysis’ ’) {}
        void init() {...}
        void analyze() {...}
        void finalize() {...}
    private:
        ...
    };
    DECLARE_RIVET_PLUGIN(Name_Of_Analysis);
}
```

Here the ellipses denote omitted detail. What is shown above is the backbone of a Rivet analysis, so let’s start building on top of this. We should note here that the correct header files to include at the top of the file can be found from the Rivet documentation online (<http://rivet.hepforge.org/trac/wiki>). We will however briefly discuss some of the main header files and classes in this mini-report.

In the next section we show an example where we have written a simple Rivet code which places a veto on events which have $E_T < 60$ GeV, and outputs the value of E_T to a histogram called MET.

B.2. Toy Analysis

```
#include "Rivet/Analysis.hh" //Required in all analyses
#include "Rivet/RivetAIDA.hh" //for Histograms
#include "Rivet/Projections/FinalState.hh" //for FinalState
#include "Rivet/Projections/MissingMomentum.hh" //for
VisibleFinalState

namespace Rivet {
  class TOY_ANALYSIS : public Analysis {
  public:
    TOY_ANALYSIS() : Analysis("TOY_ANALYSIS") {}
    void init() {
      const FinalState fs(-5, 5, 0*GeV);
      addProjection(fs, "FS");
      // for pTmiss
      addProjection(VisibleFinalState(-4.9,4.9),"vfs");
      _hist_MET = bookHistogram1D("METHisto10GeVBins", 20, 0, 200);
    }
    void analyze(const Event& event) {
      const double weight = event.weight();
      ParticleVector vfs_particles =
        applyProjection<VisibleFinalState>(event, "vfs").particles();
      FourMomentum pTmiss;
      foreach ( const Particle & p, vfs_particles ) {
        pTmiss -= p.momentum();
      }
      double eTmiss = pTmiss.pT();
      if(eTmiss/GeV<60) {
        vetoEvent;
      }
      _hist_MET->fill(fabs(eTmiss/GeV), weight);
    }
    void finalize() {
      normalize(_hist_MET, 1/sumOfWeights());
    }
  private:
    AIDA::IHistogram1D* _hist_MET;
  };
  DECLARE_RIVET_PLUGIN(TOY_ANALYSIS);
}
```

C. Understanding Toy Analysis - Initialize

As the included header files and general set-up of the Rivet code are relatively self-explanatory and can also be understood very simply from reading example codes, we move on to discuss the main functions in the Rivet code. The first of these is *Initialize* or *void init()* as it is written in the code. The *init()* function is where we tell Rivet which *projections* (i.e functions, calculators ..) we would like to access in our analysis. For instance, in the case of TOY ANALYSIS we wish to loop over all visible final state particles and take the negative sum of their momenta. Hence, we must first add a projection for accessing the final state, which is done in the following lines:

```
const FinalState fs(-5, 5, 0*GeV);
    addProjection(fs, 'FS');
```

The syntax here means that we will only consider particles in a pseudorapidity range $-5 < \eta < 5$ and that we would like to place a basic p_T cut on final state objects of 0 GeV (not very necessary, but this is only an example!).

Now, since we wish to act on *visible* final state particles (i.e those seen by a detector - so no neutrinos!) then we must build further upon this projection, and this is what we do in the next line:

```
addProjection(VisibleFinalState(-4.9,4.9), 'vfs');
```

i.e we have told Rivet that we wish to define a projection which will allow us to access the visible final state of each event. The *VisibleFinalState(x,y)* command is a set command in Rivet and has the syntax *VisibleFinalState(η_{min} , η_{max})*. Remember that we have already placed a minimum p_T cut on all particles in the event so we do not need to do so here.

The final command in the *init()* function in TOY ANALYSIS is simply an instance of us booking a histogram which we would like to fill during the analysis:

```
_hist_MET = bookHistogram1D('METHisto10GeVBins', 20, 0, 200);
```

In this case the syntax reads

```
histogram_name = bookHistogram1D('Name_of_histogram_in_ROOT', #bins,
x_min, x_max)}
```

D. Understanding Toy Analysis - Analyze

When we define the *analyze* function in a Rivet code, you must supply the following parameters to the arguments:

```
void analyze(const Event& Something_to_call_your_event) {...}
```

Where the parameter *Something_to_call_your_event* is chosen by the user (but in the majority of Rivet codes this is chosen to be *event*). Whenever you wish to access the event information you will do so via this parameter. An example of this is found in the next line of the code when we wish to extract the event weight and thus write:

```
const double weight = event.weight();
```

Which we can see makes use of Rivet's *weight()* function. Now, in order for us to use the projections we set up in initialize, we must 'apply' the projection. This essentially amounts to us defining a name for the projection which can be passed to 'foreach(...)' loops in the Rivet code (these are the constructs through which we can loop over elements of an event - to be discussed later). We apply the projection through the following command:

```
ParticleVector vfs_particles =
    applyProjection<VisibleFinalState>(event, "vfs").particles();
```

Here it is important to note the first part of the command, *ParticleVector*. This is specific to the type of projection you are applying, so the best place to find out which corresponds to which projection is to check the online Rivet documentation and class lists (<https://rivet.hepforge.org/code/dev/annotated.html>). We have then defined a name for the projection *vfs_particles* which again we will use later for accessing the projection information. The following part of the command

```
applyProjection<VisibleFinalState>(event, "vfs").particles();
```

is just standard Rivet syntax, where we must include Rivet's standard name for the projection - in this case *VisibleFinalState*. For another example, if we were applying Rivet's projection for charged final states we would have a command of the form:

```
const ChargedFinalState& charged =
    applyProjection<ChargedFinalState>(event, "CFS");
```

Similar commands apply to the projection for jets and the other projections available in Rivet. The letters in quotation marks "vfs" must match up with the name you gave to the projection in the void *init()* function.

Now onto some physics. The next few lines of the code are almost self explanatory but we will comment here on a few important details:

```
FourMomentum pTmiss;
    foreach ( const Particle & p, vfs_particles ) {
        pTmiss -= p.momentum();
    }
    double eTmiss = pTmiss.pT();
    if (eTmiss/GeV<60) {
        vetoEvent;
    }
```


Here we can see that it is possible to define a vector `pTmiss` which we then fill with the negative sum of the momenta of all the visible final state particles. The *foreach(...)* loop is the tool we use in Rivet in order to scan over all the particles which we wish to study. Since we are using the *Particle* class defined in Rivet we must supply this as an argument to the foreach loop and give a name by which we can access the particle information, here given by `p`. This will then mean we can easily grab whatever information about the particle we would like later in the loop, for example:

```
p.momentum().pT();
p.momentum().eta();
p.momentum().Et();
```

will find the particle's transverse momentum, pseudorapidity and transverse energy respectively. These and many more functions can of course be found in the online Rivet documentation. As you can see from the piece of code above, we have used this in TOY ANALYSIS in order to calculate the missing transverse energy in the event. The last thing to note about this section is that when Rivet gets information such as p_T , it does so with the units attached. This is why we have to divide by GeV when we implement our E_T cut in the next few lines:

```
if (eTmiss/GeV<60) {
    vetoEvent;
}
```

We should also note here that the *vetoEvent* command can be used at any point in the code, and very conveniently discards the current event and causes the `analyze()` function to proceed to the next event.

The final part of the `analyze` function in TOY ANALYSIS is simply the filling of the histogram we defined in `void init()`:

```
_hist_MET->fill ( fabs ( eTmiss/GeV ) , weight );
```

The syntax again is very simple - we give the name of the histogram, supply the name of the quantity we wish to fill it with and then also include the event weight, which Rivet gets from your MC generator. If you wished to fill this histogram without using the event weights then you would instead simply write:

```
_hist_MET->fill ( fabs ( eTmiss/GeV ) );
```

E. Understanding Toy Analysis - Finalize and Completing the Code

The `void finalize()` function is where we normalize any histograms we have filled earlier in the Rivet code. The command for normalizing a histogram can take two forms, the

first we have shown in the TOY ANALYSIS code and is written like the following:

```
normalize(_hist_MET, 1/sumOfWeights());
```

The syntax here being

```
normalize(Name_of_histogram, area/sumOfWeights());
```

where *area* is the numerical value of the area you wish your histogram to be normalized to and `sumOfWeights()` is the sum of all your event weights. The `sumOfWeights()` variable is predefined in Rivet so there is no need to calculate it.

After the void `finalize()` function there is a small section where we tell AIDA (the histogramming program) the name of all the histograms defined in our code. This section for TOY ANALYSIS is shown below:

```
private:
    AIDA::IHistogram1D* _hist_MET;
\end{lstlisting}
```

If we have more than one histogram then we simply list all of the histograms we have defined in this section one after another, for example:

```
\begin{lstlisting}
private:
    AIDA::IHistogram1D* _hist_MET;
    AIDA::IHistogram1D* _hist_pT;
    AIDA::IHistogram1D* _hist_eta;
```

if we had two more histograms called `_hist_pT` and `_hist_eta` respectively.

The last thing you must do in any Rivet code is declare the name of your analysis for Rivet's plugin system. The name you specify must match up to the name of the file which the code is in and also the name you have given to your analysis before your initialize function (see the TOY ANALYSIS code above).

```
DECLARE_RIVET_PLUGIN(TOY_ANALYSIS);
```

This then means that your code can be compiled with Rivet.

F. Conclusions

In this mini-report we have studied a simple example Rivet analysis code written by the author. It is hoped that by discussing some of the details of this code that readers will feel confident enough to begin writing their own analysis codes in Rivet. This report was very brief however, and when learning how to code there is no substitute for looking at examples. The codes for the analyses listed in the Rivet manual (<http://rivet.hepforge.org/>)

provide a great source of information especially pertaining to implementing more complex physics studies into Rivet.