NAIL

Natural Analysis Implementation Language

Andrea Rizzi, University and INFN of Pisa

Outline

~~The need for an analysis language~~

The current prototype

The plan ahead

NAIL: Natural Analysis Implementation Language

The current prototype of NAIL allows to express the event processing operations in declarative form, removing the need for an event loop and for explicit loops on collections of objects

- NAIL is not a full DSL, but rather is an embedded-DSL (thanks Jim for the definition!)
- NAIL is written in python and has a python interface
- Currently using ROOT RDataFrame as a backend (but loosely coupled to it)
- Allow to express new variable definitions with C++ code snippets (or calls to C++ libraries)

In short it retains python flexibility on the interface side (e.g. to easily manipulate strings, dicts etc..) but keep the code definition in C++ to be fast.

The current output of NAIL is a C++ program to be compiled and run OR a C++ library loadable with ROOT even in a python environment

The final goal of NAIL is not to only define the “event processing” in a declarative way, but the full analysis including how to build signal/background models from set of histograms and define statistical intepretation

NAIL syntax

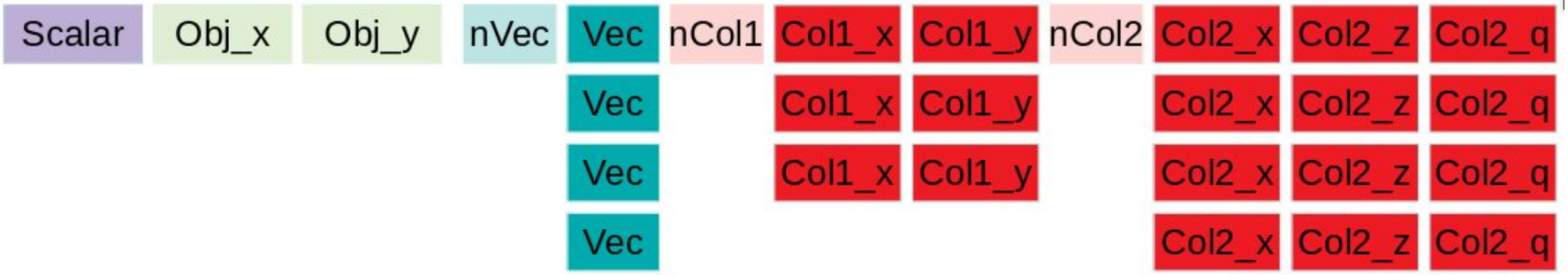
The syntax resembles very much RDataFrame “Defines”, so what does it add?

- The possibility to view “objects” as a whole in a completely transparent way
  - No need to define which input columns belong to an object, it is just based on naming rules (default naming rule is CMS NANOAOD naming convention, i.e. Collection_attributeName)
- Select SubCollections of objects
- Define new properties for objects, not necessarily scalars (e.g. a simple macro exist to build a LorentzVector back from individual _pt,_eta,_phi,_mass attributes)
- Define operations among collections, e.g.:
  - Combinations
  - DeltaR matching
- Filter events using some particular selection

Assumed data model

For LHC experiment a typical data model for NTuples is the one used in NANOAOD, the following different kind of “columns” are defined:

- Scalars (e.g. event number, Sum_et)
- Group of scalars representing a single object (e.g. MET_pt, MET_phi)
- Collections of scalars
- Collections of objects (saved in “split mode” as independent vectors of properties with a common length)



Example syntax

```
1  from nail import *
2  import ROOT
3  import sys
4
5
6  flow=SampleProcessing("ttbar","80CCFAD3-FF1A-0D43-BBBE-09278343E0EB.root")
7  #flow=SampleProcessing("ttbar","/scratch/arizzi/TTbar.root")
8  # Toy Analysis for dileptonic and semileptonic ttbar
9  # * Loose Lepton selection (requires pt>20, a  Loosflag, relative isolation < 0.25)
10 #    * Both electrons and muons
11 #Muons
12 flow.DefaultConfig(muIsoCut=0.13,muIdCut=0,muPtCut=25)
13 flow.Define("Muon_id","Muon_tightId*3+Muon_softId")
14 flow.Define("Muon_iso","Muon_miniPFRelIso_all")
15 flow.SubCollection("LooseMuon","Muon",sel="Muon_iso < muIsoCut && Muon_id > muIdCut && Muon_pt > muPtCut")
16 flow.Define("LooseMuon_p4","@p4v(LooseMuon)")
17
18 #Electrons
19 flow.Define("Electron_p4","@p4v(Electron)")
20 flow.DefaultConfig(eIsoCut=0.13,eIdCut=3,ePtCut=25)
21 flow.Define("Electron_id","Electron_cutBased")
22 flow.Define("Electron_iso","Electron_miniPFRelIso_all")
23 flow.SubCollection("LooseEle","Electron",sel="Electron_iso < eIsoCut && Electron_id > eIdCut && Electron_pt > ePtCut")
```

This just expands to 3 defines

This has an implicit loop on all Muons

This is just aliasing

This is like defining LooseMuon_pt, LooseMuon_iso, LooseMuon_anything

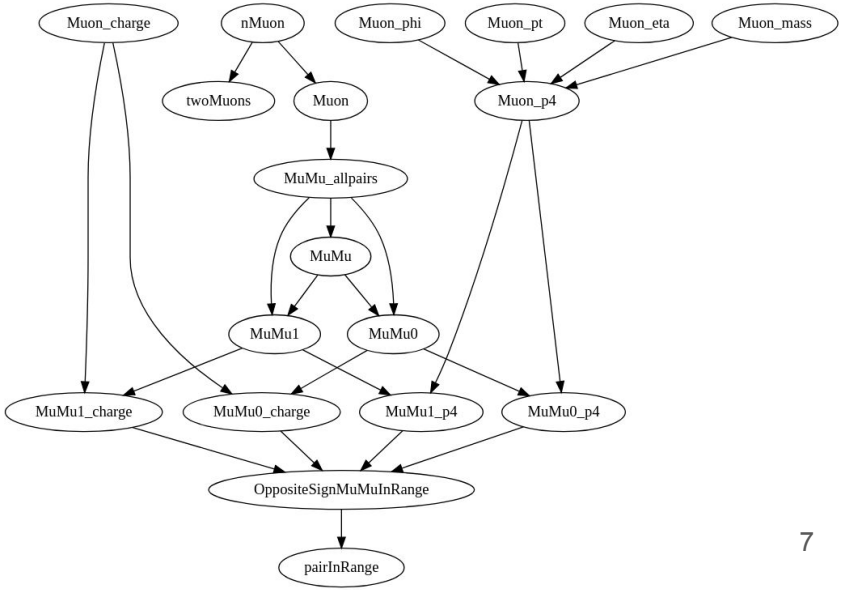
This define LorentzVector for all LooseMuons

NAIL: the computational graph

Full computational graph with all dependencies available just from the list of definitions to allow optimizations, version checking, caching, on demand / lazy calculations, etc..

Default cuts can also be defined as any other variable so that their variation/optimization can be implemented transparently

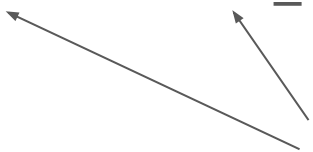
```
flow.Define("Muon_p4","@p4v(Muon)")
flow.Selection("twoMuons","nMuon>=2")
flow.Distinct("MuMu","Muon")
flow.Define("OppositeSignMuMuInRange",'''Nonzero(MuMu0_charge != MuMu1_charge
           && abs(MemberMap(MuMu0_p4+MuMu1_p4,M())-90) < 30 )''',
           requires=["twoMuons"])
flow.Selection("pairInRange","OppositeSignMuMuInRange.size() > 0")

histosPerSelection={
"pairInRange" : ["MET_pt"],
}

flow.binningRules = [
(".*_pt","100,0,500"),
]
flow.printRDFCpp([],debug=False,outname="tmp.C",selections=histosPerSelection)
```



Autodetect inputs while writing C++ code

Re-implementing a nice feature of ROOT::RDataFrame:

Define(“Muon_id”, “Muon_tightId*3+Muon_softId”)



Automatically detect that among the C++ identifiers in this C++ snippet there are 2 that are known input variables or known “defines” from earlier declarations

More complex syntax

```
26  #Lepton collection
27  flow.Define("LooseMuon_pid","(LooseMuon_pt*0)+13")
28  flow.Define("LooseEle_pid","(LooseEle_pt*0)+11")
29  flow.MergeCollections("Lepton",["LooseMuon","LooseEle"])
30
31  #Match jets to selected loose Leptons
32  flow.Define("Jet_p4","@p4v(Jet)")
33  flow.MatchDeltaR("Jet","Lepton",embed=(["pt","pid"],[]))
34
35  # * Jet select/cleaning against loose leptons , jet pt > 25 , jet id
36  flow.DefaultConfig(jetPtCut=25,jetIdCut=0,jetPUIdCut=0)
37  flow.SubCollection("CleanJet","Jet",'''
38   Jet_pt > jetPtCut &&
39   Jet_jetId > jetIdCut &&
40   Jet_puId > jetPUIdCut &&
41   (Jet_LeptonIdx==-1 || Jet_LeptonDr > 0.3)
42  ''')
```

Proper “Broadcast” operation to be defined (this is an hack)

Merging Electron and Leptons in a single “Lepton” collection.
All common attributes are available
E.g. “pid” was defined right before to distinguish e from mu

Creates:
Lepton_JetDr
Lepton_JetIdx
Jet_LeptonDr
Jet_LeptonIdx
Jet_LeptonPt
Jet_LeptonPid

Even more complex syntax

```
# * Find opposite sign, same flavour pairs
flow.Selection("twoLeptons","nLepton>=2")
flow.Distinct("LPair","Lepton")
flow.Define("isOSSF","LPair0_charge != LPair1_charge && LPair0_pid == LPair1_pid",requires=["twoLeptons"])
flow.Selection("hasOSSF","Sum(isOSSF) > 0")
#    * Closest to Z
flow.TakePair("ZLep","Lepton","LPair","Argmax(-abs(MemberMap((LPair0_p4+LPair1_p4),M() )-91.2)*isOSSF)",requires=["hasOSSF"])
flow.Define("Z","ZLep0_p4+ZLep1_p4")
flow.Define("Z_mass","Z.M()")
#    * Highest pt pair
flow.TakePair("LepHPt","Lepton","LPair","Argmax((MemberMap((LPair0_p4+LPa
flow.Define("OSSFHPt","LepHpt0_p4+LepHPt1_p4")
flow.Define("OSSFHPt_mass","OSSFHPt.M()")

#    * Highest mass pair
flow.TakePair("LepHM","Lepton","LPair","Argmax(MemberMap((LPair0_p4+LPair1_p4),M() )*isOSSF)",requires=["hasOSSF"])
flow.Define("OSSFHM","LepHM0_p4+LepHM1_p4")
flow.Define("OSSFHM_mass","OSSFHM.M()")
```

Create a selection just checking at nLeptons

Pairs of leptons

Some definitions make sense only in some region of the phase space.
The “requires” flag explicitly allow to define such dependency.
The requirements are inherited by any other quantity that depends on this one
This allows checks to be performed later

The “requirements” syntax

While adding “requires” to a variable Definition is optional, debugging RDF crashes because access to muon_pt[1] is out of boundaries is much harder to debug than having a printout saying

```
Cannot make histogram for Z_mass in OneLeptonControlRegion because the following selections are needed: twoMuons, twoOppositeSignMuons
```

Weights

Weights play a special role in physics analysis

- We use them in (N)NLO MC samples
- We use them to correct data/MC disagreements, to reweight pileup etc…
- We use weight variations for some systematic shifts

NAIL handles two types of weights:

- CentralWeights => weights to be used “as default” in a given Selection (i.e. in a phase space region)
- VariationWeights => sets of systematic variations

Variations of individual nodes
For systematics or optimizations

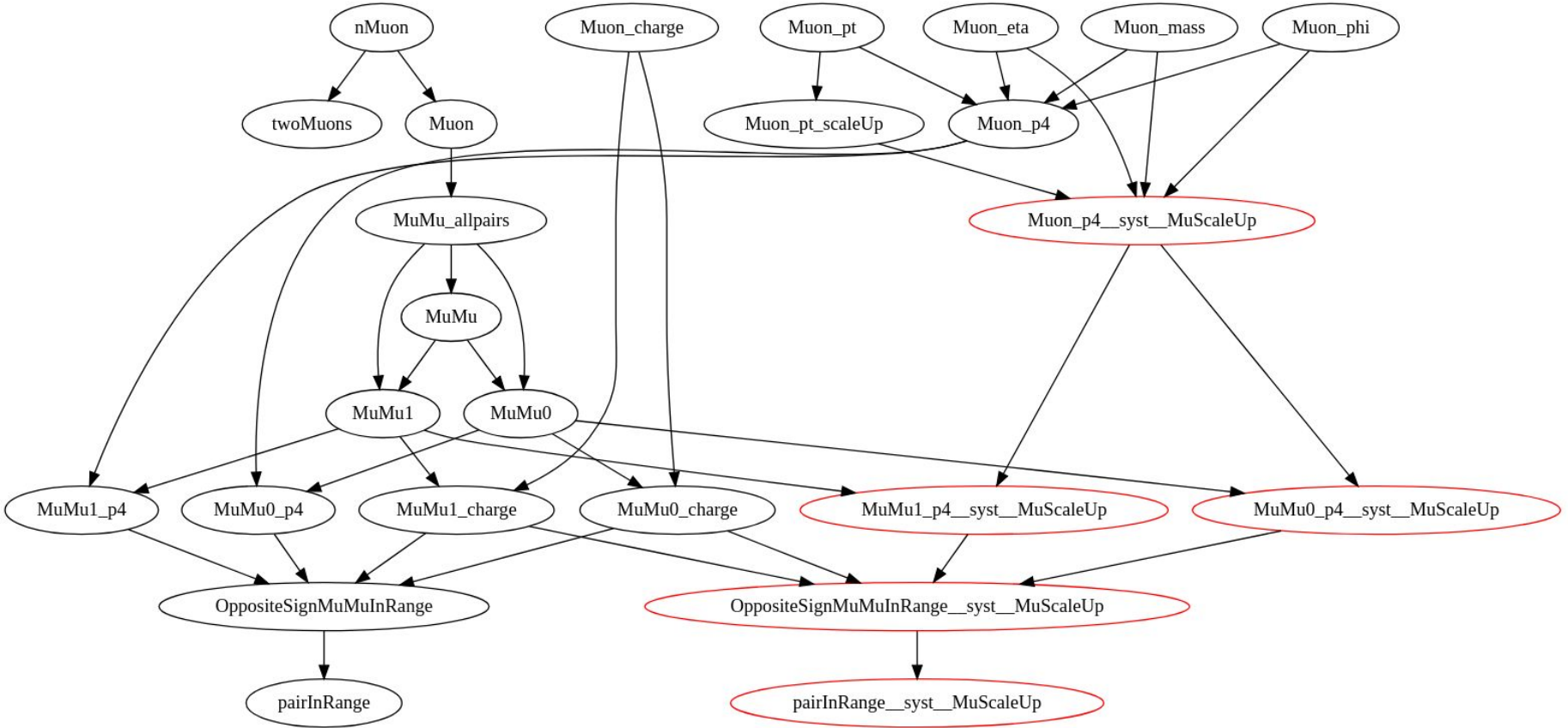
A different class of systematics, wrt to those handled with weights, are the “variations of the inputs” (e.g. Muon_pt varied with scale uncertainties)

```
#Define Systematic variations
flow.Define("Muon_pt_scaleUp","Muon_pt*1.01f") #this should be protected against systematic variations
flow.Define("Muon_pt_scaleDown","Muon_pt*0.97f")
flow.Systematic("MuScaleDown","Muon_pt","Muon_pt_scaleDown") #name, target, replacement
flow.Systematic("MuScaleUp","Muon_pt","Muon_pt_scaleUp") #name, target, replacement
```

Then simply call *createSystematicBranches*

- Only the nodes of the computational graph that are affected by a systematic are duplicated
- Nothing is “forgotten” because the dependency detection is automatic (an explicit protection against variations can be coded)
- NAIL automatically avoids to create systematic variations of sytematic variations… (e.g. Weight systematics are not used on quantities that are already originating from input-value-change systematic shifts and viceversa)

Graph with added systematic (look for “__syst__”)



Current way to run

- Currently the simplest way to test and run is to generate a C++ program using RDF backend, but many other interfaces are possible (a python library to integrated into a PyROOT RDF session has also being developed)
- The generated program contains only code that is actually needed
  - I.e. the generation of the code takes as argument the output nodes/histograms/variables to store in ntuples that you care for
  - Code generation (and even code snippets parsing!) is completely lazy
- As RDF is the only current backend, MultiThreading is available

List of operations/functions

To define the computation graph (i.e. the embedded-DSL keywords):

- DefaultConfig
- Define
- Selection
- SubCollection
- SubCollectionFromIndices
- Distinct & TakePair/Triplet
- ObjectAt
- MergeCollections
- MatchDeltaR
- CentralWeight
- VariationWeight
- VariationWeightArray

To modify the graph or access graph properties:

- createVariationBranches
- allNodesTo
- allNodesFrom
- findAffectedNodesForVariationOnTargets
- printRDFCpp
- Inputs
- Requirements

The HSF benchmarks/examples

Thanks to Gordon (and others) a set of examples and benchmarks have been defined to compare “how would you do this” in different languages and fwks.

See discussion here:

https://indico.cern.ch/event/813207/contributions/3412136/attachments/1837377/3010922/go

Their implementation for nail is available at:

https://github.com/arizzi/nail/tree/master/benchmarks

… it would be nice to have those implemented for other DSL discussed in this workshop!

E.g. benchmark 8

“In events with >=3 leptons and a same-flavour opposite-sign lepton pair, find the best same-flavour opposite-sign lepton pair (mass closest to 91.2 GeV), and plot the transverse mass of the missing energy and the leading other lepton [something whose formulation in an imperative language is easy, but whose translations to a functional language may be less clear and/or possibly inefficient]” (G. Petrucciani proposal)

```
flow.Define("Muon_p4","@p4v(Muon)")
flow.Define("Electron_p4","@p4v(Electron)")
flow.Define("Electron_pid","Electron_pt*0+11")
flow.Define("Muon_pid","Muon_pt*0+13")
flow.MergeCollections("Lepton",["Muon","Electron"])
flow.Define("Lepton_index","Range(nLepton)")
flow.Distinct("LPair","Lepton")
flow.Selection("twoLeptons","nLepton>=2")
flow.Define("isOSSF","LPair0_charge != LPair1_charge && LPair0_pid == LPair1_pid",requires=["twoLeptons"])
flow.Selection("hasOSSF","Sum(isOSSF) > 0")
flow.TakePair("Z","Lepton","LPair","Argmax(-abs(MemberMap((LPair0_p4+LPair1_p4),M() )*isOSSF-91.2))",requires=["hasOSSF"])
flow.Selection("threeLeptons","nLepton>=3",requires=["twoLeptons","hasOSSF"])
flow.SubCollection("ResidualLeptons","Lepton",sel="Lepton_index != LPair0[Z_indices] && Lepton_index != LPair1[Z_indices]",r
flow.ObjectAt("ResidualLepton","ResidualLeptons","Argmax(ResidualLeptons_pt)")
flow.Define("MET_eta","0.f")
flow.Define("MET_mass","0.f")
flow.Define("MET_p4","@p4(MET)")
flow.Define("METplusLepton_p4","ResidualLepton_p4+MET_p4")
flow.Define("METplusLepton_Mt","METplusLepton_p4.Mt()")

histosPerSelection={
"threeLeptons" : ["METplusLepton_Mt"]
}

flow.binningRules = [
(".*Mt.*","100,0,1000")
]
```

Conclusions/Plan ahead

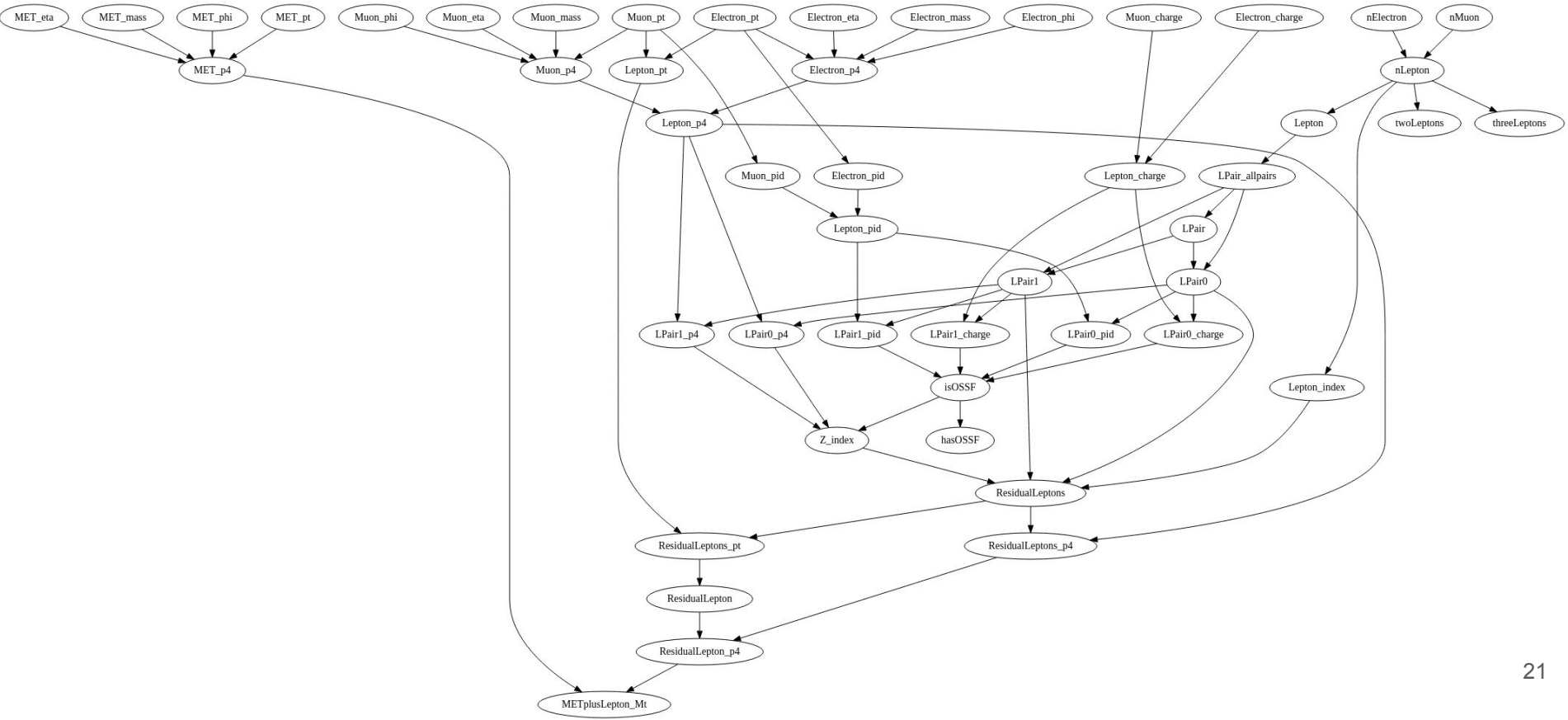
- NAIL is a prototype started 4 months ago, not a very mature project!
- It is usable to generate RDF code and make plots/derived ntuples

Plans:

- Consolidate and clean-up the interface (but as use case are still being understood I'll keep changing it for a while)
- Implement A to Z CMS analysis with it
- Allow to express operations on aggregated data and stat interpretations
- Introduce automatic/optimized caching of intermediate results
- Introduce ML interface
- Once a full featured fwk is in place, clean/refactor/optimize the code

Backup

Benchmark 8 graph



How we analyze data today

Centrally handle data processing for “established” procedures

- Event generation, simulation, reconstruction, object ID, etc...
- Consolidate procedures developing event formats closer to analysis level (MiniAOD 2013, NanoAOD 2017)

Final steps of analyses currently include

- Apply some level of corrections/systematic variations to measured quantities
- Handle multiple datasets
- Compute derived quantities event by event (invariant masses, MVAs etc..)
- Accumulated over events (i.e. make histograms)
- Build signal and background models and actual data observations from histograms (possibly with some subtractions, rescaling and other operations on the histogrammed data)
- Perform (multiple hypothesis) statistical interpretations

Analysis tools and frameworks

Most groups have shared tools and frameworks to perform the analysis tasks, but…

- Corrections and variations recipes are always handcrafted or should be frequently updated (often with cut&paste from twikis or someone's github)
- Handling of multiple datasets sometimes requires tricky slicing/splitting/merging of differently preselected processes (DY+Njets, DY pt bins, DY HT bins)
- Derived quantities are coded in "macros" that perform explicit event loops with no possibility for optimization
- Accumulated informations are needed also for syst variations and are often obtained just "rerunning" everything

On the bright side:

- Most frameworks/tools have "config files" that allow to express easily things such as the variable to histogram, the histogram binnings, the dataset splitting
- At least for statistical interpretation the "combine" tool provided a common language (in CMS)

Towards Declarative Analysis Languages

Try to express the “what” of the “analysis flow” without expressing the “how”

I.e. the analysis language should look more like what you write in the paper than what you write in C++ code

- No implementation details
- No explicit looping or cut&paste repeated code (not on events, not on collections of objects, not on data samples, not on systematic variations, not on optimization scans)
- Allow to give definitions of new quantities and abstract operations between quantities
- Ideally cover everything from event processing to statistical interpretation
    - Focus area (current prototype): event processing
    - Other areas benefit already of declarative approach in most of the existing toolsets