Stony Brook University

Writing a new fitter

Cristóvão Vilela

FiTQun! Workshop
TRIUMF

August 18 2015

Introduction

- The fiTQun class contains:
  - Fundamental detector/physics/code parameters, like the refractive index, attenuation length, quantum efficiency, quiet flag, …
  - Low-level building blocks like the observed time and charge arrays, predicted charge arrays, NLL arrays. …
  - Mid-level methods like the one-ring charge prediction, vertex pre-fitter, time peak search, …
  - High-level fitters like the π^0 fitter, the multi-segment muon fitter, the sequential multi-ring fitter, …
- The higher-level fitters are complex, and would benefit from living in their own classes
- The fiTQun class is already bloated, and as we continue to develop more complex fitters the situation might get worse

What to do

1) Fundamental parameters that will be required by all fitters are moved to fiTQun_shared, or in some cases are made `static`
   - A singleton class, acessible to all
2) This frees up the fiTQun class so that it can be inherited by other classes, without running into unset variable problems
3) New fitters inherit from fiTQun, so they can make use of the low- and mid-level fiTQun building blocks

As Gabriel was writing the p → K **ν** fitter, we implemented 1 – 3

4) Move high-level fitters that already exist in fiTQun out into their own classes

How to write a new fitter

- A version of fiTQun where the fundamental variables were moved from fiTQun to fiTQun_shared can be found in:
  - https://github.com/fiTQun/fiTQun/tree/feature/MoveRTparsToShared
- Once the changes are fully validated and everyone is happy with them, we can move them into the develop branch
  - More on these branches later today
- Gabriel built his $p \rightarrow K \boldsymbol{\nu}$ fitter off of this branch

- The next slides are a concrete example based on Gabriel's work towards writing the $p \rightarrow K \boldsymbol{\nu}$ fitter
  - All the code snippets are his
  - https://github.com/gsantucci/fiTQun/tree/feature/PDK_muGamma

Step 1: Start a new class

Inheriting from fiTQun

Static wrappers required for `Minuit`

`DoFit` to be called in runfiTQun.cc

`RunMinimizer` sets initial parameters and runs Minuit

Calculate the Negative Log Likelihood

```
#include "fiTQun.h"

class PDK_muGamma: public fiTQun {

 private:
  bool same3Vertex;
  static PDK_muGamma* static_PDK_muGamma;

 public:
  PDK_muGamma();
  ~PDK_muGamma();
  double DoFit(int ievent_, int itwnd_, int seedtype, int fittype);
  bool GetSame3Vertex() {return same3Vertex;}

 protected:
  double RunMinimizer(double *X, int& PCflg, int *flgparfix, double *arStps=NULL);
  double SetPDK_muGammaSeed(double* fitpars, int& PCflg);
  static void   PDK_muGammaWrapper(int& nDim, double* gout, double& result, double par[], int flg);
  void   SaveTestPDK_muGammaFit(double* fitparams, int istage, double totmu, double nll, int PCflg);
  double PDK_muGammaGetTwoRngnglogL(double *X, int& PCflg);
  void   GetTruePDK_muGammaParams(double* tParams);
  double muGammaNglogL(double *X, int& PCflg);
};
```

Step 2: Write a NLL function

```
double PDK_muGamma::muGammaNglogL(double *X, int& PCflg){

  Resetmu();
  //      0   1   2   3     4      5    6   7   8   9   10   11     12    13
  // X = {X1, Y1, Z1, T1, theta1, phi1, p1, X2, Y2, Z2, T2, theta2, phi2, p2}

  double X1st[8];
  for (int i=0; i<=6; i++) X1st[i]=X[i];
  X1st[7] = 0.;

  double X2nd[8];
  for (int i=0; i<=6; i++) X2nd[i]=X[i+7];
  X2nd[7] = 0.;

  if (same3Vertex) {
    for (int i=0; i<=2; i++) X2nd[i]=X[i];
  }

  int PCflg1=fiTQun::OneRing(imu,X1st,0);
  int PCflg2=fiTQun::OneRing(ie,X2nd,1);
  PCflg = abs(PCflg1)+2*abs(PCflg2);
  if ((PCflg1<0)||(PCflg2<0)) PCflg = -1;

  double nll = fiTQun::nglogL();

  return nll;

}
```

Make use of inherited fiTQun core functionality

Step 3: Write a static NLL wrapper

- Minuit requires FCN to be a static method

```
void PDK_muGamma::PDK_muGammaWrapper(int& nDim, double* gout, double& result, double par[], int flg){

  double partmp[fiTQun_shared::nPDK_muGammaParams];
  for (int i=0; i<fiTQun_shared::nPDK_muGammaParams; i++)
    partmp[i]=par[i];

  double lnLpnl=0.;

  if (par[6]<fiTQun_shared::Get()->pCherenkovThr(imu)) {
    lnLpnl+=pow((fiTQun_shared::Get()->pCherenkovThr(imu)-par[6]),2.)*10.;
  }

  if (par[13]<fiTQun_shared::Get()->pCherenkovThr(ie)) {
    lnLpnl+=pow((fiTQun_shared::Get()->pCherenkovThr(ie)-par[13]),2.)*10.;
  }

  int PCdumflg;

  result = static_PDK_muGamma->muGammaNglogL(partmp,PCdumflg) + lnLpnl;
}
```

Write NLL in terms of explicitly static methods

```
PDK_muGamma::PDK_muGamma() {
  std::cout << "Starting PDK - Knu Fitter." << std::endl;
  nring=2;
  static_PDK_muGamma = this;
  //double tParams[14];
  //GetTruePDK_muGammaParams(tParams);
  //  fqshared = fiTQun_shared::Get(-1);
}
```

Step 4: Write a RunMinimizer method

```
double PDK_muGamma::RunMinimizer(double *X, int& PCflg, int *flgparfix, double *arStps){

  int nPar = fiTQun_shared::nPDK_muGammaParams;
  //      0   1   2   3     4       5    6   7   8   9   10   11     12    13
  // X = {X1, Y1, Z1, T1, theta1, phi1, p1, X2, Y2, Z2, T2, theta2, phi2, p2}
```

1) new TFitter (nPar)
2) SetFCN (myStaticNLL)
3) SetParameters – Start, step size, limits
4) Run minimizer
5) Check result
6) delete TFitter
7) return NLL

Step 5: Write DoFit

```
double PDK_muGamma::DoFit( int ievent_, int itwnd_, int seedtype, int fittype ) {

  ievent = ievent_;
  itwnd = itwnd_;
```

1) Set the fit seed
   - For example, from a single ring fit result
2) Set fixed parameters
3) Run minimizer
4) Save result to output

Step 6: Writing result to disk

"Save" method in fiTQun_shared

```
void fiTQun_shared::SaveDefaultPDK_muGammaFit(double* fitparams, int index, double totmu, double nll, int PCflg){

  TVector3 tmppos = TfmDetToGlblCoord(TVector3(fitparams),true,false);
  tmppos.GetXYZ(fitqunpmg_.fqpmgpos1[index]);

  fitqunpmg_.fqpmgt01[index] = fitparams[3];

  TVector3 tmppos2 = TfmDetToGlblCoord(TVector3(&(fitparams[7])),true,false);
  tmppos2.GetXYZ(fitqunpmg_.fqpmgpos2[index]);

  fitqunpmg_.fqpmgt02[index] = fitparams[10];
```

Add variables to fQROOTOut

```
trout->Branch("fqpmgpcflg" , fitqunpmg_.fqpmgpcflg, "fqpmgpcflg[2]/I");
trout->Branch("fqpmgmom1" , fitqunpmg_.fqpmgmom1, "fqpmgmom1[2]/F");
trout->Branch("fqpmgmom2" , fitqunpmg_.fqpmgmom2, "fqpmgmom2[2]/F");
trout->Branch("fqpmgt01" , fitqunpmg_.fqpmgt01, "fqpmgt01[2]/F");
trout->Branch("fqpmgt02" , fitqunpmg_.fqpmgt02, "fqpmgt02[2]/F");
trout->Branch("fqpmgtotmu" , fitqunpmg_.fqpmgtotmu, "fqpmgtotmu[2]/F");
trout->Branch("fqpmgnll" , fitqunpmg_.fqpmgnll, "fqpmgnll[2]/F");
trout->Branch("fqpmgpos1" , fitqunpmg_.fqpmgpos1, "fqpmgpos1[2][3]/F");
trout->Branch("fqpmgpos2" , fitqunpmg_.fqpmgpos2, "fqpmgpos2[2][3]/F");
trout->Branch("fqpmgdir1" , fitqunpmg_.fqpmgdir1, "fqpmgdir1[2][3]/F");
trout->Branch("fqpmgdir2" , fitqunpmg_.fqpmgdir2, "fqpmgdir2[2][3]/F");
```

and fitqunout.h

```
c  proton decay to mu gamma fit information

      integer fqpmgpcflg
      real fqpmgmom1
      real fqpmgmom2
      real fqpmgt01
      real fqpmgt02
      real fqpmgtotmu
      real fqpmgnll
      real fqpmgpos1
      real fqpmgpos2
      real fqpmgdir1
      real fqpmgdir2

      common /fitqunpmg/ fqpmgpcflg(2)
     + , fqpmgmom1(2), fqpmgmom2(2), fqpmgt01(2), fqpmgt02(2)
     + , fqpmgtotmu(2), fqpmgnll(2)
     + , fqpmgpos1(3,2), fqpmgpos2(3,2), fqpmgdir1(3,2), fqpmgdir2(3,2)
```

Some work required to streamline this...

Ideas for the future

- This implementation of fitters is a step in the right direction, but clearly there are a few places to improve on
- A few ideas, for discussion:
  - Having a well defined set of methods fitters need to have implemented, so that they might be organised in some kind of "event looper" in the future
    - E.g. Init(), DoFit(), Save(), Delete(), …
  - Getting each fitter class to read its own parameters file, so that the main one can be de-cluttered
  - Writing an output wrapper. Ideally this should inherit from a template, with respect to which the rest of the code is written, so that both zbs and root output is supported
    - Then new variables, can be added by calling wrapper methods from within the fitter classes
  - Writing a template "Input" class (like WCSimWrapper, but generic) from which WCSimWrapper, and a new SKDETSIMWrapper inherit, so that either or both can be compiled depending on libraries available

Summary

- We have a way of writing new, complex, fitters which avoids further fiTQun.cc bloat
  - It might need some polishing, but it is working
- Ideally we would like to move most of the high-level stuff out of the fiTQun class and into their own classes, inheriting from fiTQun
- Other areas of fiTQun (e.g., I/O) would benefit from further factorisation
  - Need ideas/discussion