xFitter 2.2: Status and development plans

Ivan Novikov
JINR
19 March, 2019

Overview

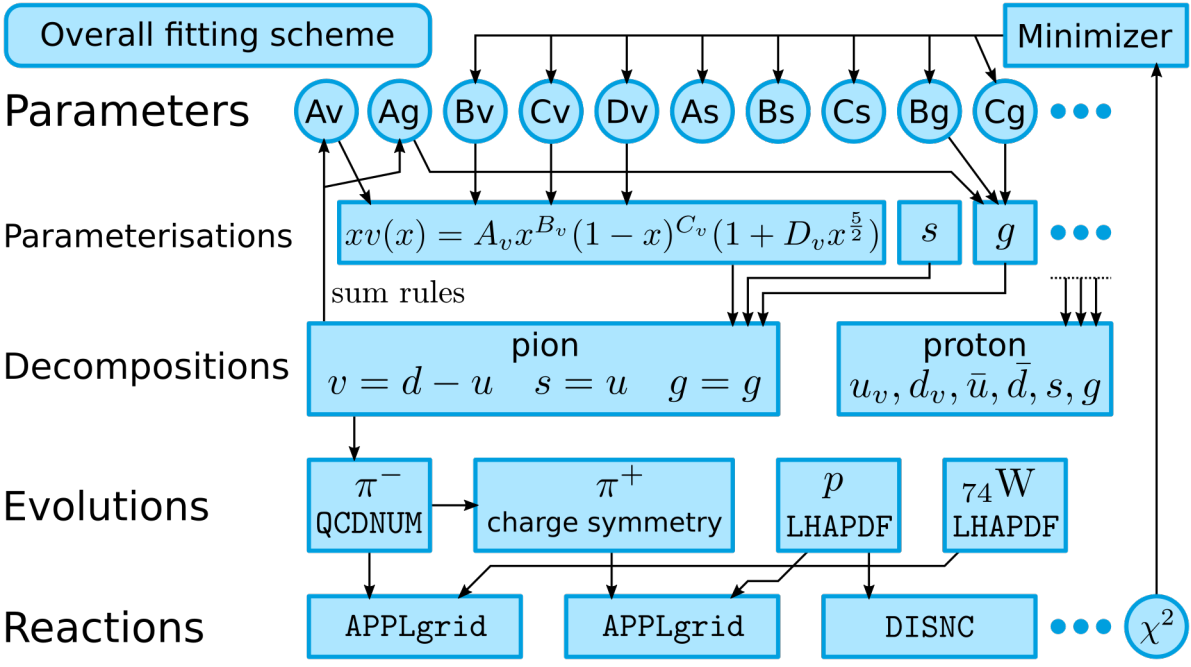
Latest developments on experimental branch `test_ceres`:
New features

- New flexible system of parameters, parameterisations, decompositions, evolutions; enabling studies of pion, kaon and nuclear PDFs.
- Parameterisation can be provided as a formula.
- Profiler: systematic shifts from uncertainties of physical constants
- Partial support for an alternative minimizer `CERES`

Plans and goals before release

- Rewrite interface of current system of reaction modules to make it work better with the new system of evolutions.
- Rewrite system of parameters to allow fitting any used physical constant or parameter. Allow using different values of parameters for each dataset or each reaction type.
- Fix features broken by the new interface. Test and verify correctness.

New features


Overall fitting scheme
Minimizer
Parameters
Av
Ag
Bv
Cv
Dv
As
Bs
Cs
Bg
Cg
Parameterisations
$xv(x) = A_v x^{B_v}(1-x)^{C_v}(1+D_v x^{\frac{5}{2}})$
s
g
sum rules
Decompositions
pion
$v = d - u \quad s = u \quad g = g$
proton
$u_v, d_v, \bar{u}, \bar{d}, s, g$
Evolutions
π^-
QCDNUM
π^+
charge symmetry
p
LHAPDF
$_{74}\mathrm{W}$
LHAPDF
Reactions
APPLgrid
APPLgrid
DISNC
χ^2

Parameter

A parameter is a variable that can be varied by the minimizer.

```
Parameters:
  #parameterName: [value, step, min, max, priorMean, priorSigma]
  Bv: [ 0.79    ,0.05]
  Cv: [ 0.37    ,0.12]
  Av:  SUMRULE #will be determined from sum rules
  Dv: [-0.86    ,0.08]
  As: [ 7.3     ,2.2 ]
  Bs: [ 1.39    ,0.16]
  Cs: [ 8       ,0] #step=0 means fixed
  Ag:  SUMRULE
  Bg: #alternative explicit syntax
    value: 6.27
    step: 1.32
    min: 0
    max: 100
    #pr_mean:
    #pr_sigma:
  Cg: [ 5       ,0]
  _0: [ 0       ,0]
  CgExpr: "=Cg^2+Bg+5" #parameter is a function of some other parameters
```

Parameterisation

Input

- Current values of parameters

Output

- Value of parameterisation function at given point x
- n-th moment $\int_0^1 xf(x)x^n \mathrm{d}x$

```
Parameterisations:
  v:
    class: Expression
    parameters: "Av*x^Bv*(1-x)^Cv*(1+Dv*x^2.5)"
  S:
    class: HERAPDF #A*x^B*(1-x)^C*(1+D*x+E*x^2+...)
    parameters: [As,Bs,Cs]
  g:
    class: PolySqrt #A*x^B*(1-x)^C*(1+D*sqrt(x)+E*x+...)
    parameters: [Ag,Bg,Cg]
```

Decomposition

Input

- Parameterisations for a given particle

Responsibilities

- Provide PDFs at initial scale in flavor basis $(d, \bar{d}, u, \bar{u}, s, \bar{s}, g, \ldots)$
- Enforce sum rules by setting normalisation parameters

```
Decompositions:
  pion:
    class: SU3_Pion
    valence: v
    sea: S
    gluon: g
  #proton: #Many decompositions at once possible
  #  class: UvDvubardbars
  #  xuv: par_uv
  #  xdv: par_dv
  #  xubar: par_ubar
  #  xdbar: par_dbar
  #  xs: par_s
  #  xg: par_g
```

Evolution

Input

- Starting scale Q_0, coupling $\alpha_s(Q_0)$, quark masses...
- PDFs at starting scale, from decomposition

Responsibilities

- Evolve PDFs to all scales, at each iteration
- Provide PDFs $xf_i(x, Q)$, $\alpha_s(Q)$ and any other evolved quantities

`xFitter` 2.2 currently supports DGLAP solvers `QCDNUM` and `APFEL++`

Some evolution classes, like `LHAPDF` and `FlipCharge`, do not actually solve DGLAP, but still provide $xf_i(x, Q)$ and $\alpha_s(Q)$

```
Evolutions:
  negative_pion:
    class: QCDNUM
    decomposition: pion #refers to previously defined
    xGrid  : [9.9e-7, 0.01, 0.1, 0.4, 0.7]
    xGridW : [1,       2,     4,   8,  16]
    Q2Grid : [1.,  2.05e8 ]
    Q2GridW: [1.,     1.]
    NQ2bins: 120
    NXbins : 200
    Read_QCDNUM_Tables: 1
    SplineOrder: 2
    ICheck : 0
  positive_pion:
    class: FlipCharge
    input: negative_pion
  tungsten_target:
    class: LHAPDF
    set:   nCTEQ15FullNuc_184_74
    member: 0
  proton:
    class: LHAPDF
    set:   nCTEQ15FullNuc_1_1
    member: 0
DefaultEvolution: negative_pion
```

```
&Data
  Name='E615-0'
  IndexDataset=2010
  Reaction='PionOnTungsten'

  TheoryType='expression'
  TermName  ='A1'
  TermType  ='reaction'
  TermSource='APPLgrid'
  TermInfo  =
    'GridName=pathToGridFile.root
    :evolution1=negative_pion
    :evolution2=tungsten_target' !<-Where to take PDFs from
  TheorExpr ='A1'
```

Currently one can only give evolutions to `APPLgrid`, other reactions always use default evolution

Profiler

Build correlated errors from uncertainties in parameters and PDFs from LHAPDF, by varying them and recording changes in predictions

```
Q0: 1.378404875209022 #initial scale =sqrt(1.9)
Order: NLO
NFlavour: 5
? !include constants.yaml
alphas: 0.118

Profiler:
  alphas: [ 0.118, 0.119, 0.117 ]
  Evolutions:
    tungsten_target:
      sets:    [nCTEQ15FullNuc_184_74]
      members: [[0,1,end]]
```

The next steps

ReactionTheory

ReactionTheory class is a singleton for computing theory predictions.

Input

- $xf(x, Q)$ and $\alpha_s(Q)$ from evolutions
- Physical constants and scales
- ! Both PDFs and physical parameters can be different for different datasets

Output

- Computed cross-section
- (optional) Estimate of theory uncertainty, from different sources

Responsibilities

- Read input, check its sanity, compute predictions

ReactionTheory

ReactionTheory class is a singleton for computing theory predictions.

Input

- $xf(x, Q)$ and $\alpha_s(Q)$ from evolutions
- Physical constants and scales
- ! Both PDFs and physical parameters can be different for different datasets

Output

- Computed cross-section
- (optional) Estimate of theory uncertainty, from different sources

Responsibilities

- Read input, check its sanity, compute predictions

Right?

Current interface

Current interface of `ReactionTheory` is the following:

- `virtual int compute(int dataSetID, valarray<double>&val, map<string,valarray<double> >&err)`
  Compute cross-sections (`val`) and their uncertainties (`err`) for a given term. `dataSetID` is id of a term (not of a dataset). Returns 0 on success, and an error code on error. Although nobody knows what the error codes mean, and what to do with them.
- `virtual void setDatasetParameters(int dataSetID, map<string,string> parsReaction, map<string,double> parsDataset)`
  Remember parameters for a given term. `dataSetID` is id of term. `parsReaction` are term-parameters (which do not include reaction-specific parameters). `parsDataset` are dataset-specific parameters (which do not include global or reaction-specific)
  Current implementations of concrete `ReactionTheory` use a bunch of `std::maps` to remember parameters for each dataset.

Additionally, `ReactionTheory` gets some parameters via `ReactionTheory::GetParam`, which returns either a reaction-specific parameter, if it is provided, or a global parameter.

What will be printed?

```
MyReaction:
  muF: 1
  muR: 1.5
  Order: NLO
  eta_cut: "2.5"
```

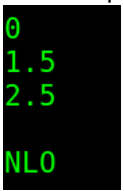
```
MyReaction::setDatasetParameters(...){
  //GetParam returns double
  cout<<GetParam("muF")<<endl;
  cout<<GetParam("muR")<<endl;
  cout<<GetParam("eta_cut")<<endl;
  //GetParamS returns string
  cout<<GetParamS("eta_cut")<<endl;
  cout<<GetParamS("Order")<<endl;
}
```

What will be printed?

```
MyReaction:
  muF: 1
  muR: 1.5
  Order: NLO
  eta_cut: "2.5"
```

```
MyReaction::setDatasetParameters(...){
  //GetParam returns double
  cout<<GetParam("muF")<<endl;
  cout<<GetParam("muR")<<endl;
  cout<<GetParam("eta_cut")<<endl;
  //GetParamS returns string
  cout<<GetParamS("eta_cut")<<endl;
  cout<<GetParamS("Order")<<endl;
}
```

Actual output

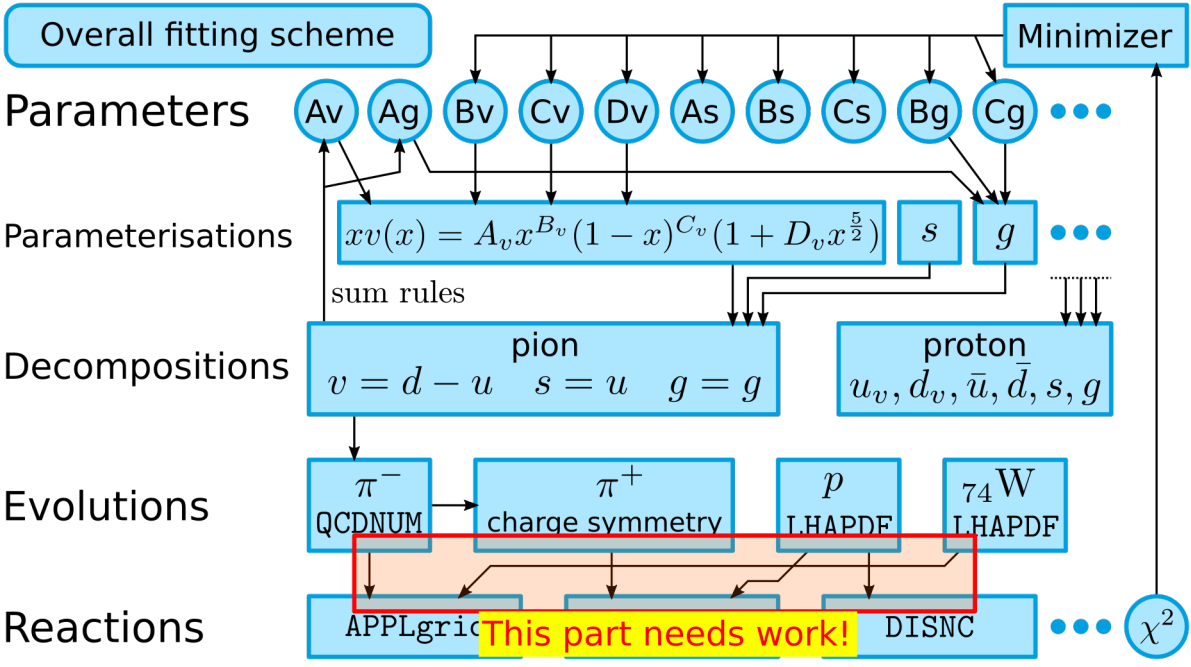


ReactionTheory

Actual responsibilities

- For each dataset, read parameters from 7 different places: global and reaction-specific (5 different functions for different data types), dataset-specific, term-specific.
- If term-specific or global string-typed parameters are provided, convert them from string to correct data type.
- If some parameter is provided in many different places, decide which parameter overwrites which. Different reactions do this differently (or not at all), there is no specified correct behavior.
- Check sanity of resulting parameters. It is not clear what happens if only some parameters are sane.
- Remember the obtained parameters for each termID.
- For each term, retrieve stored parameters by termID and compute cross-section

As far as I know, no reaction written so far can correctly handle all the various cases.


Overall fitting scheme
Minimizer
Parameters
Av
Ag
Bv
Cv
Dv
As
Bs
Cs
Bg
Cg
Parameterisations
$xv(x) = A_v x^{B_v}(1-x)^{C_v}(1+D_v x^{\frac{5}{2}})$
s
g
sum rules
Decompositions
pion
$v = d - u \quad s = u \quad g = g$
proton
$u_v, d_v, \bar{u}, \bar{d}, s, g$
Evolutions
π^-
QCDNUM
π^+
charge symmetry
p
LHAPDF
^{74}W
LHAPDF
Reactions
APPLgrid
This part needs work!
DISNC
χ^2

Design goals

- Setting and parameters come from different places and have different scope:
  1. Term-specific
  2. Dataset-specific
  3. Reaction-specific
  4. Global
  5. Defaults

  More specific definitions overshadow less specific, with priority as given here
- Input evolution(s) to `ReactionTheory` can be set separately at different scope
- Different parameter types:
  1. double
  2. int
  3. string
  4. list of double

  The system should automatically convert to type requested by `ReactionTheory`, or report an error if this is impossible
- Any double-typed parameter can potentially be fitted
- Details of all this shall be invisible to `ReactionTheory`

Main idea

```
void MyReaction::compute(TermData*td,valarray<double>&val,map<string,valarray<double> >&err){
  double muF=*td->getParam("muF");
  double muR=*td->getParam("muR");
  int nloops=td->getParam("nloops");
  string Order=td->getParam("Order");

  int verbosity;
  if(td->getParam("verbosity").isNone())
    verbosity=0;//use default value
  else
    verbosity=td->getParam("verbosity");

  BaseEvolution*evolution=td->getEvolution();
  ...
  double xf=evolution->xfxQ2(x,Q2);
  ...
  compute cross-sections, write it into val
  ...
}
```

`TermData` is a class that provides an interface to all data that is necessary to compute predictions for this term.
`ReactionTheory` gets all the input it needs through this class.

A note on pointers

We want to be able to fit any double-typed parameters.
This means that such parameters can change from iteration to iteration.
Therefore, an implementation of `ReactionTheory` must re-read parameter values each time.
It should not be assumed that they are the same.

To emphasize this important point, and to reduce the necessary number of calls to `TermData::getParam`, we decided that all double-typed parameters will be provided by pointer:

```
double *p_muR=td->getParam("muR");//returns double*, not double
double muR=*p_muR;
```

The pointer points to some location in memory where minimizer keeps current value of the parameter.
The pointer itself never changes and always points to the same location.

TermData class

Each instance of `TermData` corresponds to one reaction term.

Responsibilities

- Provide a convenient interface to parameters for `ReactionTheory`
- Get parameter definitions from different places and decide which overshadow which in a uniform manner.
- Retrieve evolutions and provide them to `ReactionTheory`
- \+ Hold additional reaction data for this term (later slide)
- \+ Manage wrappers for alternative ways of accessing PDFs (later slide)

Any and all instances of `TermData` are managed by `TheorEval`

Special parameters `"evolution"`, `"evolution1"`, `"evolution2"` are used to specify which evolution should be used for this term. Evolutions are identified by name.

TermData class

```
class TermData{
public:
  TermData(unsigned id,ReactionTheory*);
  const unsigned id;
  ReactionTheory*reaction;
  Variant getParam(string)const;
  void*reactionData=nullptr;//see later slide
  void actualizeWrappers();//see later slide
  BaseEvolution*getEvolution()const;//returns first evolution
  BaseEvolution*getEvolution(int i)const;
  //any other fields or getters that could be useful to ReactionTheory
private:
  //depends on implementation
}
```

`Variant` is a special class that automatically converts to any necessary type (`double*`, `int`, `string`,...). It can also be `None` if a requested parameter is undefined.

Variant class

```
Variant v;
cout<<v<<endl;//None
v="1234321";
cout<<v.isNone()<<endl;//false
cout<<v.type()<<endl;//String
int i=v;
cout<<i<<endl;//1234321
double d=3.1415;
v=&d;
cout<<v.isDoublePtr()<<endl;//true
cout<<v<<endl;//0x7ffd4d739ce8
cout<<*v<<endl;//3.1415
```

Reaction data

`TermData` has an additional pointer `reactionData` that `ReactionTheory` can use to store some data for this term.

```
void MyReaction::initTerm(TermData*td){
  ...
  pre-compute some quantities a and b for this term
  ...
  MyDataStruct*data=new MyDataStruct();
  data->a=a;
  data->b=b;
  td->reactionData=(void*)data;
}
void MyReaction::compute(TermData*td){
  MyDataStruct*data=(MyDataStruct*)td->reactionData;
  double a=data->a;
  double b=data->b;
  ...
}
void MyReaction::freeTerm(TermData*td){
  delete td->reactionData->a;
  delete td->reactionData;
}
```

This is more elegant than using a series of `std::maps`, as is currently done.

PDF wrappers

Some libraries require PDF input as a function:

```
void xfxQ(const double&x,const double&Q,double*results)
```

We provide wrappers like this to be used by `ReactionTheory`. However, one needs to make sure that at any given moment these wrappers are wrapping the correct PDF. This is done by calling `TermData::actualizeWrappers`.

ReactionTheory interface

Concrete implementations of ReactionTheory can override the following methods:

```
void ReactionTheory::atStart()//called once when all objects are initialized
void ReactionTheory::atIteration()//called in the beginning of each iteration
void ReactionTheory::initTerm(TermData*)//called once for each term at start
void ReactionTheory::compute(TermData*,valarray<double>&val,map<string,valarray<double> >&err)
void ReactionTheory::updateTerm(TermData*)//called when parameters for this term have changed
void ReactionTheory::freeTerm(TermData*)//called once for each term at end
void ReactionTheory::atMakeErrorBands(TermData*)
```

Of all these methods only compute must be implemented; all the others are optional.

Conclusions

`xFitter` 2.2 brings many new features and enables more complicated fits with its new flexible fitting scheme. However, more work needs to be done before release:

- Implement the new `ReactionTheory` interface, as described in this presentation
- Reimplement reaction modules using the new interface
- Reimplement missing parameterisations, evolutions etc. (Chebyshev polynomials, `apfelgrid` etc)
- Fix current issues with the build system
- Test and verify

We welcome comments, suggestions and use cases.

An informal writeup on development plan presented here is available on `xFitter` wiki on gitlab