overview of the B2DXFitters package

Manuel Schiller

CERN

July 9th-10th, 2015

introduction

- B2DXFitters is a rather versatile package
  - can do sophisticated mass/PID fits to extract yields/sWeights
    see Agnieszka's talk(s) and her part of the hands-on session
  - also does rather complicated time fits
    see my talk(s) and my part of the hands-on session
- a lot of effort has gone into making (time) fits fast
  - very necessary with ever increasing data sets and fit complexities
  - would like to transfer lessons learned to next generation...

outline

- package structure
- start with overview of time fitting part (outline of topics later)
- then hand over quickly to Agnieszka's "mass fit news" talk because I run out of time
- we can and will come back to these slides during the hands-on session, I promise. . . :)

package structure

package structure

package structure

- important to understand package structure (subdirectories):

B2DXFitters	header files for C++ algorithms
cmt	used for cmt (building)
data	various data files (templates), config files
dict	ROOT dictionaries (reflection information)
doc	release.notes, other documentation
python	reusable python code
scripts	fitting (python) scripts
src	C++ sources
standalone	standalone build dir (symlinks to src/*.cxx)
tutorial	material for hands-on session (feel free to add!)

- looking for RooFit classes: B2DXFitters, src (, standalone)
- looking for reusable parts of fit: python/B2DXFitters
- concrete fit implementations: scripts

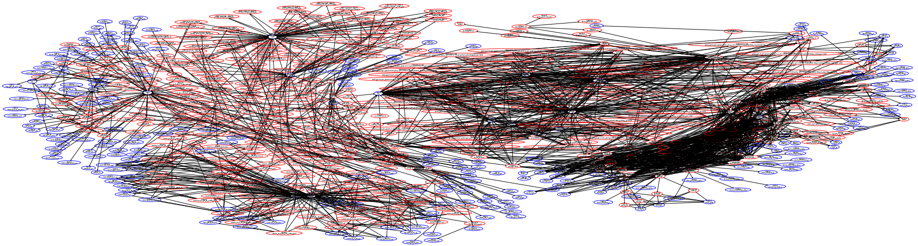
news for the time fitting part

news for the time fitting part

news for the time fitting part

- substantial code refactoring
  - reusable parts are now packaged in a reusable manner
  - should make it easy to write your own fit
- substantial improvements to documentation of routines
- brand new example scripts as tutorials for the hands-on session
- accompanying slides ($\mathcal{O}(60)$) explaining the “why” and “how”

introduction time fit

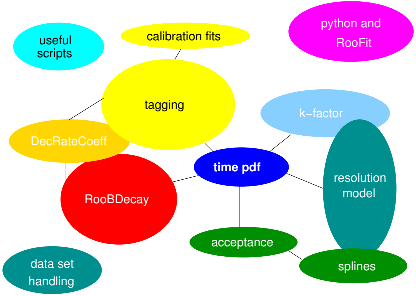


PDF structure of the 1 fb^{-1} $B_s^0 \to D_s^{\mp} K^{\pm}$ cFit

introduction time fit

- time fits are complicated beasts
  (785 pdf components for $B_s^0 \to D_s^{\mp} K^{\pm}$)
- did a lot of work to make things a little easier to use

- outline
  - philosophy
  - in-depth topics:



  - Friday: hands-on, getting started with B2DXFitters time fits

philosophy

philosophy

philosophy

- time fits should be *configurable*
  - have python dictionaries to configure the fit (high level config)
  - time fit itself should consist of building blocks that can be *reused*
- want easy interoperability for different fits
  - flexible on input side (data tuples, templates, ...)
  - rigorous on output side (predictable variable names, pdf structure)

→ pdf building should be done by program, not cut and paste!

- conceptually, a fit should look like this:

```
# get mass pdf per mode
masspdfs, yields = {}, {}
for mode in config['Modes']: # 'Bs2DsK', 'Bs2DsPi', ...
        masspdfs[mode], yields[mode] = readMassModeFromMDFit(config, ws, mode)
# construct time pdfs
timepdfs = { }
for mode in config['Modes']:
        timepdfs[mode] = buildBDecayTimePdf(config, ws, mode, ...)
# zip them together
bits = RooArgList()
for mode in config['Modes']:
    tmp = WS(ws, RooProdPdf('%s_pdf' % mode, '%s_pdf' % mode,
        timepdfs[mode], masspdfs[mode]))
    tmp = WS(ws, RooExtendPdf('%s_epdf' % mode, '%s_epdf' % mode,
        tmp, yields[mode]))
    bits.add(tmp)
totpdf = RooAddPdf('totpdf', 'totpdf', bits)
```

- will illustrate reusable building blocks on the next few slides

buildBDecayTimePdf

- most of the hard work is actually done by a single routine: `buildBDecayTimePdf`
- let's have a look at its signature:

```
def buildBDecayTimePdf(
    config,                             # configuration dictionary
    name,                               # 'Signal', 'DsPi', ...
    ws,                                 # RooWorkspace into which to put the PDF
    time, timeerr, qt, qf, mistag, tageff,      # potential observables
    Gamma, DeltaGamma, DeltaM,          # decay parameters
    C, D, Dbar, S, Sbar,                # CP parameters
    timeresmodel = None,                # decay time resolution model
    acceptance = None,                  # acceptance function
    timeerrpdf = None,                  # pdf for per event time error
    mistagpdf = None,                   # pdf for per event mistag
    mistagobs = None,                   # real mistag observable
    kfactorpdf = None,                  # distribution k factor smearing
    kvar = None,                        # variable k which to integrate out
    aprod = None,                       # production asymmetry
    adet = None,                        # detection asymmetry
    atageff = None                      # asymmetry in tagging efficiency
    ):
    # ...
```

- you can do pretty much anything with it!
- will use hands-on to move from a simple fit (average η, σ_t) to something a lot more complicated
- in practise, you'll need to know what this "magic" routine does (roughly)

RooBDecay

RooBDecay

RooBDecay

- we all know and love `RooBDecay`:

$$P(t) \sim e^{-\Gamma t} \cdot \quad \left(A \cdot \cosh(\tfrac{\Delta\Gamma}{2}) + B \cdot \sinh(\tfrac{\Delta\Gamma}{2}) + C \cdot \cos(\tfrac{\Delta m}{2}) + D \cdot \sin(\tfrac{\Delta m}{2})\right)$$

- good building block for fast fit:
  - analytical time integral (normalisation!)
  - analytical convolution with resolution models (gaussian(s))
- physics is usually encoded in A, B, C, D
- however, slows down if
  - we have per-event observables (σ_t, η): need to normalise on every event (e.g. $A, B, C, D \to (A, B, C, D)(\eta, P(\eta))$, so need normalising!)
  - we have an acceptance: can normalise $P(t)$ analytically, but not $P(t) \cdot a(t)$ or $P(t') \otimes G(t - t') \cdot a(t)$

$\rightarrow$ will show how these slowdowns can be overcome

acceptance

acceptance

acceptance

- problem: no analytical normalisation of $P(t) \cdot a(t)$ or $P(t') \otimes G(t-t') \cdot a(t)$ in general case
- two solutions:
  - approximation: bin $a(t)$

$$\int dt\, P(t)\, a(t) \to \sum_i a(t_i) \int_{bin\, i} dt\, P(t)$$

  - approximation: $a(t)$ piecewise polynomial $\to$ splines!
    - acceptance becomes part of resolution model:

$$G(t-t') \to G_a(t-t')$$

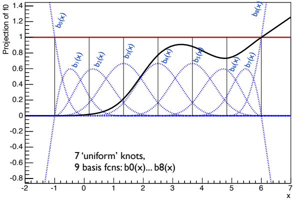
    - normalisation of convolution integral can be done analytically

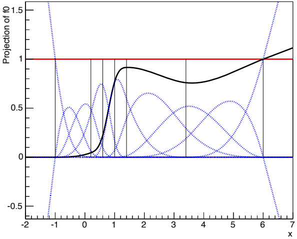
splines: introduction

- splines are low order polynomials which approximate the function on a "subrange", e.g. $p(x) = a + bx + cx^2 + dx^3$
- you have many subranges which compose interval over which function is to be approximated: $p_0(x), p_1(x), \ldots$
- typically, you want
  - $p_i(x_i) = y_i$
  - $p_i'(x_i) = p_{i-1}'(x_i)$
  - $p_i''(x_i) = p_{i-1}''(x_i)$

spines: introduction

- idea much clearer with Gerhard's excellent picture:





Splines

- Piece-wise (cubic) polynomials
- Parameterized by 'knots' (interval boundaries) and values at these knots
  - can of course have uniform and non-uniform intervals
- It can be proven that a cubic spline is the answer to the question: amongst all twice differential smooth functions that go through a set of specified points, which one is the 'stiffest' (ie. smallest average 2nd derivative) function?
- Splines can be written as a sum over 'base splines' -- eg. 'cubic b-splines'.
- For n knots, there are n+2 b-splines.
- base splines $b_i(x)$
  - ONLY depend on the knot definition
  - form a partition of unity: $\sum_i b_k(x) = 1$
    - given an efficiency, can easily create an *inefficiency*:

$$\epsilon(x) = \sum_k a_k b_k(x)$$

$$\Rightarrow 1 - \epsilon(x) = \sum_k (1 - a_k) b_k(x)$$

G. Raven, CP Tools and Techniques meeting, July 25th 2013

M. Karbach, G. Raven, M. Schiller: LHCb-INT-2013-043

Thursday, July 25, 13

8

1D splines as acceptance

- define knots (x_i):

```
time = RooRealVar('time', 'time', 0.2, 15.)
myknots = [ .2, .4, .6, .8, 1., 2., 3., 6., 12.]
knotbinning = RooBinning(time.getMin(), time.getMax(), 'knotbinning')
for v in myknots:
    knotbinning.addBoundary(v)
knotbinning.removeBoundary(time.getMax())
knotbinning.removeBoundary(time.getMin())
```

- define spline coefficients

```
coefflist = RooArgList()
for i in xrange(0, len(knots)):
    coefflist.addRooRealVar('SplineAccCoeff%u' % i, 'SplineAccCoeff%u' % i, coeffs[i], 0., 2.))
```

- create the spline, and the resolution model from it

```
tacc = RooCubicSplineFun('SplineAcceptance', 'SplineAcceptance', time, 'knotbinning', coefflist)
fit_resmodel = RooGaussEfficiencyModel('fit_resmodel', 'fit_resmodel', time, tacc, zero, timeerr, SF, SF)
```

- that's it (essentially), can use this resolution-model-with-acceptance in RooFit classes like RooDecay, RooBDecay, ...

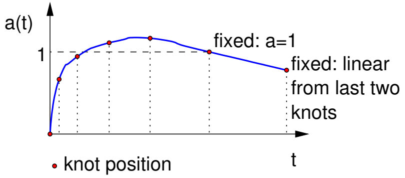
1D splines for use as acceptance

- there are a couple of stumbling stones (as usual):
  - knot intervals must fully cover the fit range, and may not leak "outside"
  - for generation, it's faster to use a `RooEffProd` of the resolution-model-convolved `RooBDecay` and the spline

    $\rightarrow$ different PDFs for generation and fitting!
    (nice cross-check!)
    - if you want to generate toys, make sure your spline coefficients are all smaller than 1

1D splines for use as acceptance

- choose knot positions (more where curvature of acceptance is high)
- can then fit spline coefficients to control channel/MC/…
- problems:
  - overall scale of spline not set:
    fix one coefficient to 1
  - at large times (low stats), tend to pick up stat. fluctuations, but expect uncurved acceptance
    → fix last knot coefficient from linear extrapolation of previous two



1D splines for use as acceptance

- naturally, this comes canned as a python routine:

```
from B2DXFitters.acceptanceutils import buildSplineAcceptance

# time range e.g. from 0.2 to 15 ps
acc, accnorm = buildSplineAcceptance(ws, time, 'Bs2DsK_acceptance',
    [ 0.5, 1.0, 1.5, 3.0, 6.0, 12.0 ], # knot positions
    [ 0., 0.5, 1.0, 1.0, 1.0, 1.0 ],   # initial coefficients (last two fixed, see last slide)
    True) # float spline coefficients in fit
```

- acc is an acceptance suitable for fitting
- acc_norm is a normalised acceptance suitable for generation with RooEffProd
  (applies overall scaling factor such that $a(t) < 1$ for all t)

$\rightarrow$ docs in python/B2DXFitters/acceptanceutils.py

spline systematics

- this is always hard, but fortunately not very hard...
- for cubic splines, approximation error will be proportional to $\frac{\partial^4 f(x)}{\partial x^4} \cdot h^4$ (h: spline subinterval size)
- no need to calculate that: just try with twice the number of knots, and get estimate from the difference
- if you're not stable, you likely have some problem with your approximation; plot to investigate

binned approximation

- occasionally still useful:
  - cross-checks
  - in other fits
- two implementations:
  - resolution-model based (just like splines - not cubic, but constant!): same use as splines, but use `RooBinnedFun` instead of `RooCubicSplineFun`
  - older implementation based on `RooEffHistProd`, binning existing function as fast approximation

```
time = ... # RooRealVar for the time
acc = ... # some acceptance function
binning = RooUniformBinning(timelo, timehi, nbins, "someNameForBinning")
time.setBinning(binning, "someNameForBinning")
binned cc= RooBinnedPdf("name", "title", time, "someNameForBinnig", acc)
finalpdf = RooEffHistProd("name", "title", pdf_wo_acc, binnedacc)
```

→ also useful to avoid numerical integration in e.g. mass fits which are sculpted by some efficiency/threshold function...

DecRateCoeff

DecRateCoeff

DecRateCoeff

- very versatile class
- includes the tagging in RooBDecay
- will therefore go slowly through the material
  - average mistag
  - per-event mistag
  - asymmetries
  - advanced: combining taggers

DecRateCoeff: basics (1/3)

- in $D_s K$, the pdf changes depending on final state charge (q_f) and tagging decision (q_t)
- for average mistag ω, the coefficient in front of e.g. the $\cos(\Delta m t)$ term is composed from contributions from B_s and $\overline{B_s}$:

$$
\begin{aligned}
C_{eff} &= \sum_{q_i \in \{B, \overline{B}\}} P(q_f, q_t | q_i) \cdot C(q_i, q_f) \\
&\sim \begin{cases}
\epsilon_{tag}(C_f(1-\omega) + C_f(-\omega)) & q_t = +1,\ q_f = +1 \\
\epsilon_{tag}(-C_f(-\omega) - C_f(1-\omega)) & q_t = -1,\ q_f = +1 \\
(1-\epsilon_{tag})(C_f - C_f) & q_t = 0,\ q_f = +1 \\
(1-\epsilon_{tag})(\overline{C}_{\bar{f}} - \overline{C}_{\bar{f}}) & q_t = 0,\ q_f = -1 \\
\epsilon_{tag}(\overline{C}_{\bar{f}}(-\omega) + \overline{C}_{\bar{f}}(1-\omega)) & q_t = -1,\ q_f = -1 \\
\epsilon_{tag}(-\overline{C}_{\bar{f}}(1-\omega) - \overline{C}_{\bar{f}}(-\omega)) & q_t = +1,\ q_f = -1
\end{cases}
\end{aligned}
$$

- make sure you recognise it as the formula we all know and love!
- it has (conceptually) a pdf inside, so it must *normalise* itself

DecRateCoeff: average mistag

- can play this game for
  - CP-odd coefficients (for the sin / cos(Δmt) terms): C enters with sign of $q_i \cdot q_f$
  - CP-even coefficients (for the sinh / cosh($\frac{\Delta\Gamma t}{2}$) terms): C enters without sign
- constructor:

```
DecRateCoeff(const char* name, const char* title, Flags flags,
        RooAbsCategory& qf, RooAbsCategory& qt,
        RooAbsReal& Cf, RooAbsReal& Cfbar,
        RooAbsReal& tageff, RooAbsReal& eta,
        RooAbsReal& aprod, RooAbsReal& adet,
        RooAbsReal& atageff);
```

- `flags` can be `CPEven` or `CPOdd`
- or `| Minus` to it when you need an overall minus sign in front of C
- if you want to fit $\overline{C} = (C_f + \overline{C}_{\bar{f}})/2$ and $\Delta C = (C_f - \overline{C}_{\bar{f}})/2$, or `| AvgDelta`
- put asymmetries to `RooConstVar("zero", "zero", 0.)` if you don't need them

DecRateCoeff: basics (2/3)

- for (calibrated) per-event mistag $\omega(\eta)$, things become a little more complicated

$$
\begin{aligned}
C_{eff} &= \sum_{q_i \in \{B, \overline{B}\}} P(q_f, q_t | q_i) \cdot P(\eta | q_t) \cdot C(q_i, q_f) \\
&\sim \begin{cases}
\epsilon_{tag} P(\eta)(C_f(1 - \omega(\eta)) + C_f(-\omega(\eta))) & q_t = +1,\ q_f = +1 \\
\epsilon_{tag} P(\eta)(-C_f(-\omega(\eta)) - C_f(1 - \omega(\eta))) & q_t = -1,\ q_f = +1 \\
(1 - \epsilon_{tag}) U(\eta)(C_f - C_f) & q_t = 0,\ q_f = +1 \\
(1 - \epsilon_{tag}) U(\eta)(\overline{C}_{\bar{f}} - \overline{C}_{\bar{f}}) & q_t = 0,\ q_f = -1 \\
\epsilon_{tag} P(\eta)(\overline{C}_{\bar{f}}(-\omega(\eta)) + \overline{C}_{\bar{f}}(1 - \omega(\eta))) & q_t = -1,\ q_f = -1 \\
\epsilon_{tag} P(\eta)(-\overline{C}_{\bar{f}}(1 - \omega(\eta)) - \overline{C}_{\bar{f}}(-\omega(\eta))) & q_t = +1,\ q_f = -1
\end{cases}
\end{aligned}
$$

- $U(\eta)$ is a uniform distribution (whatever you set the mistag to for untagged events, you'll always get the same contribution)

DecRateCoeff: per-event mistag

- same game
- constructor:

```
DecRateCoeff(const char* name, const char* title, Flags flags,
    RooAbsCategory& qf, RooAbsCategory& qt,
    RooAbsReal& Cf, RooAbsReal& Cfbar,
    RooAbsRealLValue& etaobs, RooAbsPdf& etapdf,
    RooAbsReal& tageff, RooAbsReal& eta,
    RooAbsReal& aprod, RooAbsReal& adet,
    RooAbsReal& atageff);
```

- etaobs is the observable η
- etapdf is $P(\eta)$
- eta is the calibrated mistag $\eta_c(\eta) = \omega(\eta)$

DecRateCoeff: basics (3/3)

- with asymmetries, this becomes even more complicated:
  - $\epsilon_{tag} \to \epsilon_{tag} \cdot (1 + q_t a_{tag})$
  - $\omega(\eta) \to \omega(\eta), \overline{\omega}(\eta)$
  - add factor $(1 + q_i a_{prod})$ everywhere
  - add factor $(1 + q_f a_{det})$ everywhere
- full expression too large (and ugly) for slides, so see
  - appendix to 1 fb^{-1} $D_s K$ ANA note
  - doxygen docs for `DecRateCoeff` (`make doxy` in standalone)

DecRateCoeff: asymmetries

- same game yet again
- constructor:

```
DecRateCoeff(const char* name, const char* title, Flags flags,
    RooAbsCategory& qf, RooAbsCategory& qt,
    RooAbsReal& Cf, RooAbsReal& Cfbar,
    RooAbsRealLValue& etaobs, RooAbsPdf& etapdf,
    RooAbsReal& tageff, RooAbsReal& eta, RooAbsReal& etabar,
    RooAbsReal& aprod, RooAbsReal& adet,
    RooAbsReal& atageff);
```

- etaobs is the observable η
- etapdf is $P(\eta)$
- eta is the calibrated mistag $\eta_c(\eta) = \omega(\eta)$ for B
- etabar is the calibrated mistag $\overline{\eta_c}(\eta) = \overline{\omega}(\eta)$ for $\overline{B}$

DecRateCoeff: extra: combining taggers (1/7)

- in principle, one can write down the fromulae on the past few slides for more than one tagger
  - that would (correctly) combine multiple taggers within `DecRateCoeff`
  - rewrite in progress, but not ready for production yet
- I had hoped to be faster to avoid what follows, but…
- combining taggers workaround, required steps:
  - split into three mutually exclusive taggers ($|qt| = 1$ for OS, 2 for SSK, 3 for both OS+SSK)
  - calibrate taggers as preprocessing step with average p_0, p_1, mangle tagging decision (last item)
  - run toy to propagate asymmetries and errors on calibration to calibrated mistag
  - fit with six calibrations, one for each tagger and true B flavour
  - constrain the various p_0, p_1 according to result of toys

DecRateCoeff: extra: combining taggers (2/7)

- step 1: tuple preprocessing
  - suppose you have a RooDataset with your data somewhere
  - use these steps to add two new variables to it:

```
import MistagCalibration, DLLTagCombiner, TagDLLToTagEta, TagDLLToTagDec, RooArgList

# uncalibrated mistags eta_OS, eta_SSK, decisions qt_OS, qt_SSK; there are part of data set
# calibration constants are p0_OS, p1_OS, etaavg_OS, similar for SSK
eta_OSc = WS(ws, MistagCalibration('eta_OSc', 'eta_OSc', eta_OS, p0_OS, p1_OS, etaavg_OS))
eta_SSKc = WS(ws, MistagCalibration('eta_SSKc', 'eta_SSKc', eta_SSK, p0_SSK, p1_SSK, etaavg_SSK))
qts = RooArgList(qt_OS, qt_SSK)
etas = RooArgList(eta_OSc, eta_SSKc)
dll = WS(ws, DLLTagCombiner('dlltag', 'dlltag', qts, etas))
eta = WS(ws, TagDLLToTagEta('eta', 'eta', dll))
qt = WS(ws, TagDLLToTagDec('qt', 'qt', dll, qts))

# add to data set
dataset.addColumn(eta)
dataset.addColumn(qt)
```

  - then save the data set to a new workspace
- if you prefer to work with straight tuples, have a look at `TagCombiner.h`

DecRateCoeff: extra: combining taggers (3/7)

- step 2a: work out correction for calibration asymmetries
  - in step 1, we apply average correction for $B/\overline{B}$
  - not correct yet, so correct remaining discrepancy in fit
  - use toy to figure out the post-combination calibration asymmetries (is exact for linear calibration polynomials)
  - `standalone/taggingtoy/tagcomb.cc` contains the code
    1. generates events with correct calibration for $B/\overline{B}$, and OS and SSK (mistag templates from ROOT file, just splot your tuple after mass fit)
    2. combines using average calibration for OS and SSK
    3. recalibrates output separately for $(B, \overline{B})$x(OS only, SSK only, OS+SSK)
    4. use to figure out post-combination calibrations, and correlations
  - you get six sets of calibration constants, which are all correlated among each other, see next slide

DecRateCoeff: extra: combining taggers (4/7)

- step 2b: work out correction for calibration asymmetries
- `make tagcomb; ./tagcomb` (eventually) prints

```
[... much output ...]
Total calibration:
B    OS only : eta_c = 0.376730+/-0.004389 + (1.048155+/-0.039917) * (eta - 0.371147) --- correl(p0, p1) = -0.111791
B    SSK only: eta_c = 0.404896+/-0.011412 + (0.995879+/-0.148797) * (eta - 0.414892) --- correl(p0, p1) = -0.122611
B    OS + SSK: eta_c = 0.338363+/-0.005959 + (1.027861+/-0.038725) * (eta - 0.338493) --- correl(p0, p1) = -0.874463
Bbar OS only : eta_c = 0.365517+/-0.004395 + (0.950216+/-0.040072) * (eta - 0.371147) --- correl(p0, p1) = -0.111883
Bbar SSK only: eta_c = 0.424801+/-0.011414 + (1.004340+/-0.150355) * (eta - 0.414892) --- correl(p0, p1) = -0.123455
Bbar OS + SSK: eta_c = 0.338781+/-0.006030 + (0.971845+/-0.039962) * (eta - 0.338493) --- correl(p0, p1) = -0.878334

Correlations:
          0        1        2        3        4        5        6        7        8        9       10       11
 0  1.00000 -0.11179  0.00000  0.00000  0.49565 -0.12126  0.88340 -0.09034  0.00000  0.00000  0.43630 -0.11593
 1 -0.11179  1.00000  0.00000  0.00000 -0.17072  0.36865 -0.09043  0.80830  0.00000  0.00000 -0.13861  0.30321
 2  0.00000  0.00000  1.00000 -0.12261  0.65815 -0.54123  0.00000  0.00000  0.93878 -0.12029  0.63338 -0.52537
 3  0.00000  0.00000 -0.12261  1.00000 -0.63105  0.81198  0.00000  0.00000 -0.12244  0.98640 -0.60841  0.78788
 4  0.49565 -0.17072  0.65815 -0.63105  1.00000 -0.87446  0.43682 -0.13789  0.62212 -0.62217  0.94027 -0.84147
 5 -0.12126  0.36865 -0.54123  0.81198 -0.87446  1.00000 -0.10427  0.29779 -0.51403  0.80069 -0.83010  0.95060
 6  0.88340 -0.09043  0.00000  0.00000  0.43682 -0.10427  1.00000 -0.11188  0.00000  0.00000  0.49475 -0.13419
 7 -0.09034  0.80830  0.00000  0.00000 -0.13789  0.29779 -0.11188  1.00000  0.00000  0.00000 -0.17092  0.37480
 8  0.00000  0.00000  0.93878 -0.12244  0.62212 -0.51403  0.00000  0.00000  1.00000 -0.12345  0.67273 -0.55661
 9  0.00000  0.00000 -0.12029  0.98640 -0.62217  0.80069  0.00000  0.00000 -0.12345  1.00000 -0.61649  0.79852
10  0.43630 -0.13861  0.63338 -0.60841  0.94027 -0.83010  0.49475 -0.17092  0.67273 -0.61649  1.00000 -0.87833
11 -0.11593  0.30321 -0.52537  0.78788 -0.84147  0.95060 -0.13419  0.37480 -0.55661  0.79852 -0.87833  1.00000
```

- this includes contributions from combination, stat. and syst. errors
- corresponding tables exist earlier in the output for
  - combination only
  - combination + stat. error
  - combination + syst. error
  - (total error)

DecRateCoeff: extra: combining taggers (5/7)

- step 2b: work out tagging efficiency asymmetries post combination
  - splitting OS and SSK taggers into OS only, SSK only and OS+SSK changes the tagging efficiencies and asymmetries for the three "new taggers"
  - look at `standalone/taggingtoy/eps.c` to see how to calculate it

```
> make eps; ./eps
Combining tagging efficiencies (signal):
 OS: eps = 0.387000+/-0.003000 Delta eps = -0.001970+/-0.001260
SSK: eps = 0.477000+/-0.003000 Delta eps =  0.000220+/-0.000040

 OS only: eps=0.202401+/-0.001952 a=-0.002756+/-0.001628
SSK only: eps=0.292401+/-0.002330 a= 0.001837+/-0.001029
  OS+SSK: eps=0.184599+/-0.001843 a=-0.002315+/-0.001629

Correlation:
   1.00000000e+00  -9.63105978e-01   2.49481592e-01   1.01449534e-02   7.02032244e-03   1.02339761e-02
  -9.63105978e-01   1.00000000e+00   2.03354154e-02  -8.05565545e-03  -5.77788479e-03  -8.17299797e-03
   2.49481592e-01   2.03354154e-02   1.00000000e+00   8.98034829e-03   5.01061453e-03   8.88495266e-03
   1.01449534e-02  -8.05565545e-03   8.98034829e-03   1.00000000e+00  -9.99652998e-01   9.98788285e-01
   7.02032244e-03  -5.77788479e-03   5.01061453e-03  -9.99652998e-01   1.00000000e+00  -9.97590361e-01
   1.02339764e-02  -8.17299794e-03   8.88495268e-03   9.98788284e-01  -9.97590361e-01   1.00000000e+00
```

- correlation matrix ordered $(\epsilon_{OS}, \epsilon_{SSK}, \epsilon_{OS+SSK}, a_{OS}, a_{SSK}, a_{OS+SSK})$

DecRateCoeff: extra: combining taggers (6/7)

- step 3: set up fit
  - get mistag templates for the three taggers (OS only, SSK only, OS+SSK) from the data added in step 1
  - use constants for six calibrations from step 2a
  - use tagging efficiencies and asymmetries from step 2b
  - set up constraints for calibration constants (12D) and tagging efficiencies and asymmetries (6D), see next slide
  - then, use this DecRateCoeff constructor:

```
DecRateCoeff(const char* name, const char* title, Flags flags,
    RooAbsCategory& qf, RooAbsCategory& qt,
    RooAbsReal& Cf, RooAbsReal& Cfbar,
    RooAbsRealLValue& etaobs, RooArgList& etapdfs,
    RooArgList& tageffs, RooArgList& etas, RooArgList& etabars,
    RooAbsReal& aprod, RooAbsReal& adet,
    RooArgList& atageffs);
```

  - same as before, RooArgLists are ordered OS only, SSK only, OS+SSK

DecRateCoeff: extra: combining taggers (7/7)

- what remains is to show how to construct the 6D or 12D constraints
- numerially tricky, since cov. matrices damn near singular
- special routine which can recover...

```
from B2DXFitters.GaussianConstraintBuilder import GaussianConstraintBuilder
cbuilder = GaussianConstraintBuilder(ws, {
    'GammaLb': 0.006, # constrain GammaLb to within 0.006
    # constrain S+Sbar, S-Sbar for Bd2DPi from PDG values (name: [ 'formula', [params], mean, error ])
    'Bd2DPi_avgSSbar': [ '0.5*(@0+@1)', ['Bd2DPi_S', 'Bd2DPi_Sbar'], +0.046, 0.023 ],
    'Bd2DPi_difSSbar': [ '0.5*(@0-@1)', ['Bd2DPi_S', 'Bd2DPi_Sbar'], -0.022, 0.021 ],
    'multivar_Bs2DsPiTagEffAsyms': [ # name: multivar_something
        # list of variables
        [ 'Bs2DsPi_TagEff0', 'Bs2DsPi_TagEff1', 'Bs2DsPi_TagEff2',
          'Bs2DsPi_AsymTagEff0', 'Bs2DsPi_AsymTagEff1', 'Bs2DsPi_AsymTagEff2' ],
        # errors
        [ 0.001952, 0.002330, 0.001843, 0.001628, 0.001029, 0.001629 ],
        # correlation matrix (always give full precision - only shortened here to fit on slide!)
        [ [  1.00000000e+00, -9.63105978e-01, 2.49481592e-01,  1.01449534e-02,  7.02032244e-03,  1.02339764e-02 ],
          [ -9.63105978e-01,  1.00000000e+00, 2.03354154e-02, -8.05565545e-03, -5.77788479e-03, -8.17299794e-03 ],
          [  2.49481592e-01,  2.03354154e-02, 1.00000000e+00,  8.98034829e-03,  5.01061453e-03,  8.88495268e-03 ],
          [  1.01449534e-02, -8.05565545e-03, 8.98034829e-03,  1.00000000e+00, -9.99652998e-01,  9.98788284e-01 ],
          [  7.02032244e-03, -5.77788479e-03, 5.01061453e-03, -9.99652998e-01,  1.00000000e+00, -9.97590361e-01 ],
          [  1.02339764e-02, -8.17299794e-03, 8.88495268e-03,  9.98788284e-01, -9.97590361e-01,  1.00000000e+00 ], ],
      ]
      # any other constraints you may have
  })
# get RooArgSet for use with fitTo's RooFit.ExternalConstraints option
constraints = cbuilder.getSetOfConstraints()
```

- very useful to handle all your constraint needs (config dictionary!)

resolution model

resolution model

resolution model, k-factors

- three big subtopics:
  - obtaining resolution model
  - considerations for a fast fit
  - k-factors (partially reconstructed/misIDed modes)

obtaining a resolution model

- easy, here are examples:

```
from B2DXFitters.resmodelutils import getResolutionModel
config = {
    'DecayTimeResolutionModel': 'GaussianWithPEDTE',
    'DecayTimeResolutionBias': 0.,              # if there is a shift
    'DecayTimeResolutionScaleFactor': 1.15,     # usually the errors need a bit of scaling
    'Acceptance': 'Spline',                     # has to work closely with spline acceptance classes
    'Context':  'GEN' # or 'FIT', as the case may be
    }
# time is decay time variable, timeerr is decay time error

# get spline acceptance from somewhere
acc = #...
resmodel, acc = getResolutionModel(ws, config, time, timeerr, acc)
```

- when you need an average decay time, use

```
config = {
    'DecayTimeResolutionModel': {
        'sigmas': [sigma_1, sigma_2, ..., sigma_N],
        'fractions': [f_1, f_2, ..., f_N-1 ] } # non-recursive, i.e. add up to 100 %
    }
```

- you can use `scripts/AvgResModel.py` to fit the widths and fractions from a decay time error distribution

→ see docs in `python/B2DXFitters/resmodelutils.py`

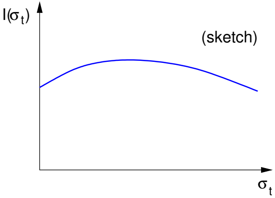
considerations for a fast fit (1/2)

- with per-event time errors, we get

$$P(t') \otimes G(t - t'|\sigma_t) \cdot P(\sigma_t)$$

$\rightarrow$ recalculation of normalisation $I(\sigma_t) = \int dt\, P(t') \otimes G(t - t'|\sigma_t)$ for every single event!

- despite analytical normalisation of convolution integral: SLOOOOOW!
- however, $I(\sigma_t)$ varies slowly with σ_t



$\rightarrow$ can tabulate in 100 points, and interpolate in between (fast!)

considerations for a fast fit (2/2)

- need to tell RooFit to use the interpolation trick:

```
from B2DXFitters.timepdfutils import parameteriseResModelIntegrals

config = { # tell how many bins in time error we need for table that's accurate enough
    'NBinsProperTimeError': 100
    }
# make it so!
parameteriseResModelIntegrals(config, ws, timeerrpdf, timeerr, resmodel)
```

- will mostly be handled by `buildBDecayTimePdf`

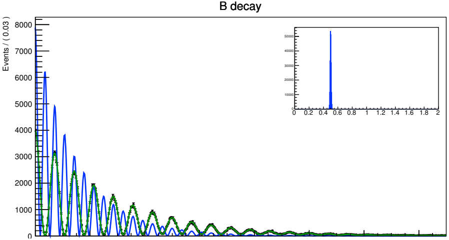
$\rightarrow$ see docs in `python/B2DXFitters/timepdfutils.py`

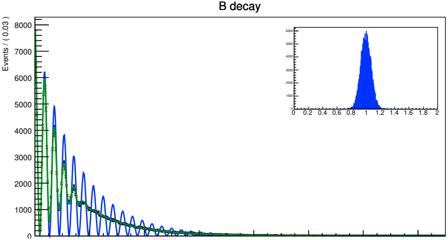
k-factors (1/3)

- lifetime is calculated along the lines of $t = |\vec{x}_{SV} - \vec{x}_{PV}| \frac{m_{B_s}}{|\vec{p}|}$
- for partially reconstructed and misid'ed modes, we get $\frac{m_{B_s}}{|\vec{p}|}$ wrong
- idea: take correction factor from MC:

$$k = \frac{(m_{B_s}/|\vec{p}|)_{true}}{(m_{B_s}/|\vec{p}|)_{reco}}$$

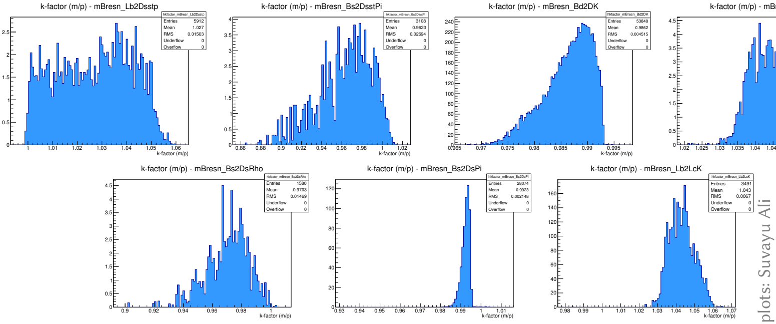
- can correct by substitution $t \longrightarrow k \cdot t$





plots: Suvayu Ali

k-factors (2/3)



- can now put this into toy generator(s) for $D_s K$
- can also use it in cFit to get the BG description correct:

$$\frac{d\Gamma}{dt}(t;\Gamma,\Delta\Gamma,\Delta m) \longrightarrow \int dk\, P(k)\cdot\frac{d\Gamma}{dt}(t;k\Gamma,k\Delta\Gamma,k\Delta m)$$

k-factors (3/3)

- convenient to express k-factor smearing as resolution model: `RooKResModel`
  - average correction for each event can be precalculated and cached

```
from B2DXFitters.timepdfutils import applyKFactorSmearing

config = {
    'NBinsTimeKFactor': 100, # use 100 bins to bin k-factor distributions
    }
# time          - decay time observable
# timeresmodel  - resolution model (after applying spline acceptance)
# kvar          - k-factor variable (unobservable!)
# kfactorpdf    - k-factor distribution for mode
# last argument: list of targets { t_i } for substitution t_i -> k * t_i

# update timeresmodel to include k-factor smearing
timeresmodel = applyKFactorSmearing(config, ws, time, timeresmodel, kvar, kfactorpdf, [ Gamma, DeltaGamma, DeltaM ])
```

- will mostly be handled by `buildBDecayTimePdf`

$\rightarrow$ see docs in `python/B2DXFitters/timepdfutils.py`

python, RooFit and ownership

python, RooFit and ownership

python, RooFit and ownership

- python's objects are *reference-counted*
  - no need for memory management
- C++/RooFit uses explicit memory management
  - need `new`/`delete`
  - ownership transfer often not clear in ROOT/RooFit (ownership: which code is responsible for calling delete)
  - worse: RooFit clones objects all over the place...

→ easy to get confused (or get python/ROOT confused)

- can you spot what's wrong with that code:

```
mean = RooRealVar('mean', 'mean', 3.4, -10., 10.)
sigma = RooRealVar('sigma', 'sigma', 1.0, 0., 5.)
x = RooRealVar('x', 'x', 0., -10., 10.)

pdf = RooGaussian('g', 'g', x, mean, sigma)

ws = RooWorkspace('ws')
ws.__getattribute__('import')(pdf)

# get data set from somewhere
pdf.fitTo(dataset)
# write pdf with fitted parameters to ROOT file
ws.writeToFile('fitresult.root')
```

python, RooFit and ownership

okay, let's go through the example slowly

- create variables and pdf – all fine so far…

```
mean = RooRealVar('mean', 'mean', 3.4, -10., 10.)
sigma = RooRealVar('sigma', 'sigma', 1.0, 0., 5.)
x = RooRealVar('x', 'x', 0., -10., 10.)

pdf = RooGaussian('g', 'g', x, mean, sigma)
```

- create and import into workspace

```
ws = RooWorkspace('ws')
ws.__getattribute__('import')(pdf)
```

! pdf, sigma, mean, x are cloned, and only the cloned versions are in the workspace!

```
# get data set from somewhere
pdf.fitTo(dataset)
```

! fit happens on original objects, not the ones in workspace

```
# write pdf with fitted parameters to ROOT file
ws.writeToFile('fitresult.root')
```

! wrote cloned objects with workspace – these have the values before the fit!

python, RooFit and ownership

- this is a nasty interaction between python and ROOT
- need to be very careful which version of the object we use!
- better: only keep one version around:

```
from B2DXFitters.WS import WS
ws = RooWorkspace('ws')

mean = WS(ws, RooRealVar('mean', 'mean', 3.4, -10., 10.))
sigma = WS(ws, RooRealVar('sigma', 'sigma', 1.0, 0., 5.))
x = WS(ws, RooRealVar('x', 'x', 0., -10., 10.))

pdf = WS(ws, RooGaussian('g', 'g', x, mean, sigma))

# get data set from somewhere
pdf.fitTo(dataset)
# write pdf with fitted parameters to ROOT file
ws.writeToFile('fitresult.root')
```

- `WS(ws, X)` imports X into ws, and returns the workspace's copy of X

→ only one copy of the object around, confusion avoided

- Use `WS(ws, X)` in python. Always.

python, RooFit and ownership

- rules of the game (to avoid leaks and crashes):
  - call `ROOT.SetMemoryPolicy(ROOT.kMemoryStrict)` (done by `import B2DXFitters`)
  - C++ objects created from within python are owned by python
    - will be freed when reference count drops to zero
    - if C++ is to take ownership, use `ROOT.SetOwnership(obj, False)`
  - objects returned from C++/ROOT routines are not owned by python
    - C++ code must call delete or similar
    - things like e.g. `pdf.createIntegral(...)` which return pointers to new (unowned) object must then call `ROOT.SetOwnership(obj, True)` in python to avoid leaks
- RooWorkspaces own all contained objects
  - don't import what you do not need
  - objects that are only needed temporarily belong in a temporary workspace

data set handling

data set handling

data sets: readDataSet (1/2)

- tuples can come from different sources
- should be possible to quickly fit with tuple of a colleague (with different branch names etc)
- example:

```
from B2DXFitters.datasetio import readDataSet

seed = 42 # it's easy to modify the filename depending on the seed number
configdict = {
    # file to read from
    'DataFileName': '/some/path/to/file/with/toy_%04d.root' % seed,
    # data set is in a workspace already
    'DataWorkSpaceName':    'FitMeToolWS',
    # name of data set inside workspace
    'DataSetNames':         'combData',
    # mapping between observables and variable name in data set
    'DataSetVarNameMapping': {
        'sample':  'sample', # phipi, kstk, kpipi, pipipi etc
        'mass':    'lab0_MassFitConsD_M',
        'pidk':    'lab1_PIDK',
        'dsmass':  'lab2_MM',
        'time':    'lab0_LifetimeFit_ctau',
        'timeerr': 'lab0_LifetimeFit_ctauErr',
        'mistag':  'tagOmegaComb',
        'qf':      'lab1_ID',
        'qt':      'tagDecComb',
        # sweights need to be combined from different branches in this
        # case, only one of the branches is ever set to a non-zero value,
        # depending on which subsample the event is in
        'weight': ('nSig_both_nonres_Evts_sw+nSig_both_phipi_Evts_sw+'
                   'nSig_both_kstk_Evts_sw+nSig_both_kpipi_Evts_sw+'
                   'nSig_both_pipipi_Evts_sw')
        }
    }
# get observables from workspace ws
obs = RooArgSet()
for obsname in config['DataSetVarNameMapping'].keys():
    obs.add(ws.obj(obsname))
# now read the data set
data = readDataSet(configdict, ws, obs)
```

data sets: `readDataSet` (2/2)

- will read data sets from RooWorkspace or flat NTuple
- sanitise input (`qf/qt` are categories, often people write doubles to tuple!)
- simple observable names in fit, irrespective of input (branch names are horrible!)
- simple formula support on input:
  - imagine people write final state charge and hasOscillated to tuple:

    ```
    'qt': 'lab1_ID*hasOscillated'
    ```
  - or s-weights come per D_s final state:

    ```
    'weight': 'sw_phipi+sw_kstk+sw_kpipi+sw_pipipi'
    ```

→ very flexible input routine!

→ docs in `python/B2DXFitters/datasetio.py`

data sets: writeDataSet (1/2)

- writing data sets to an ntuple is just as easy:

```
from B2DXFitters.datasetio import writeDataSet

data = # get RooDataSet from somewhere
writeDataSet(data,
    '/path/to/some/file.root',
    'datasetnameinrootfile',
    {
        # variable mass in data is renamed to bsmass in file, pidk is
        # uppercased
        'mass': 'bsmass', # name in data: branch name
        'pidk': 'PIDK'
        # other observables in data remain "unrenamed"
    })
```

→ docs in python/B2DXFitters/datasetio.py

templates: readTemplate1D

- reads from histogram, or RooDataSet or pdf from a workspace
- imports into given workspace, optionally "renaming" pdf observable

```
from B2DXFitters.datasetio import readTemplate1D

mistagpdf = readTemplate1D(
        'OSTagger.root',        # file name
        None,                   # None for plain histogram, or name of workspace
        'mistag',               # name of observable in file
        'hetaOS',               # histogram name in file, or name in workspace
        ws,                     # workspace into which to import
        ws.obj('mistag'),       # observable to "connect to" in ws
        'Mistag_OS_')         # prefix for imported pdf
```

- will read histo `hetaOS` from `OSTagger.root`
- creates `Mistag_OS_Pdf` in ws, which depends is `mistag`

→ docs in `python/B2DXFitters/datasetio.py`

tagging calibration

tagging calibration

tagging calibration

- needless to say, these routines can be used for tagging calibrations, too
- won't go into too much detail, but…
  - calibrations with per-event mistag are trivial:
    - simply use `MistagCalibration` class, and float p_0 and p_1
  - calibrations using tagging categories aren't much more complicated:
    - use `getMistagBinBounds` to calculate suggestions for category boundaries
    - use `getTrueOmegasPerCat` in toys to get the "right answer" for the per-category ω_i boundaries
    - use `getEtaPerCat` to calculate suggestions for category average mistags η_i (fit starting values)
    - use `fitPolynomialAnalytically` to obtain calibration parameters after the time fit has run
    - `TaggingCat` and `RooBinningPdf` classes implement tagging categories

$\rightarrow$ see docs in `python/B2DXFitters/taggingutils.py`

useful scripts

useful scripts

useful scripts (1/2)

- there are a lot of useful little helpers in B2DXFitters
- would like to introduce some:
  - from `python/B2DXFitters/utils.py`:
    - `setConstantIfSoConfigured`:
      takes list of const. parameters, and changes pdf inputs accordingly
    - `printPDFTermsOnDataSet`:
      printout of the value of each PDF component for debugging
    - `configDictFromFile`, `configDictFromString`, `updateConfigDict` to implement configuration files
  - `python/B2DXFitters/TLatexBeautifier.py`:
    rewrites simple strings like “Bs2DsK” according to simple rules, suitable as input to TLatex (plots!)

useful scripts (2/2)

- and there's more:
  - `python/B2DXFitters/FitResult.py`:
    pretty-print result, optionally blinding it[1]
  - `scripts/printFitResult.py`:
    print a fit result from a file (and unblind, if desired)
  - `scripts/make_histos.py`:
    given a set of toy results, plot pulls and residuals

[1] our blinding strategy is that we solenmly promise to never ever look at data results before unblinding; RooFit/Minuit output is disabled for data fits, and we use `FitResult` for printing (reason: RooFit's blinding mechanism "unblinds" itself when the parameter limits are close)

conclusion

conclusion

conclusion

- the B2DXFitters package can do a lot of things
  - and it is usually quite performant
- most (if not all) of the really nasty (time) PDF building is handled by `buildBDecayTimePdf`
  - to use it correctly (or debug it), you still need to have an idea of what goes on inside
  - I hope the black box has just become a little more transparent...
- I hope there was something interesting or useful for everyone!

- feel free to ask me to add whatever you feel is missing from these slides!