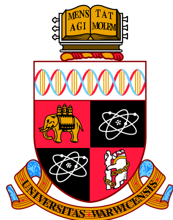
TensorFlow for amplitude analyses

Anton Poluektov

University of Warwick, UK

19 March 2018

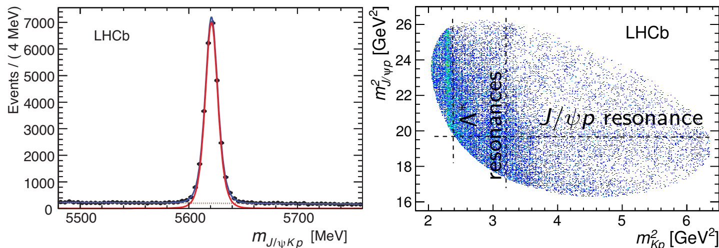


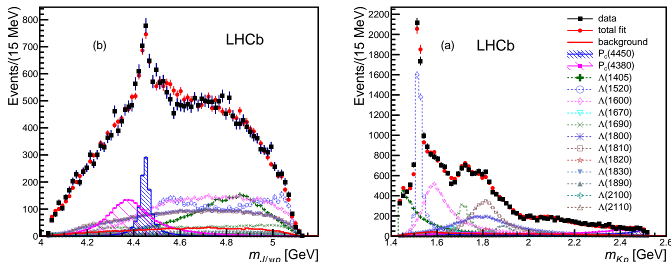
On behalf of LHCb TensorFlowAnalysis taskforce
(A. Mauri, A. Merli, A. Mathad, A. Morris, M. Martinelli, A. P.)

Introduction: amplitude analyses at LHCb

Pentaquark discovery: [PRL 115 (2015) 072001]

$\sim$ 26000 events, 6D phase space, unbinned ML fit





Introduction: maths behind amplitude analyses

Again, based on pentaquark paper: [PRL 115 (2015) 072001]

$$|\mathcal{M}|^2 = \sum_{\lambda_{\Lambda_b^0}} \sum_{\lambda_p} \sum_{\Delta\lambda_\mu} \left| \mathcal{M}^{\Lambda^*}_{\lambda_{\Lambda_b^0}, \lambda_p, \Delta\lambda_\mu} + e^{i\,\Delta\lambda_\mu \alpha_\mu} \sum_{\lambda_p^{P_c}} d^{\frac{1}{2}}_{\lambda_p^{P_c}, \lambda_p}(\theta_p)\, \mathcal{M}^{P_c}_{\lambda_{\Lambda_b^0}, \lambda_p^{P_c}, \Delta\lambda_\mu} \right|^2,$$

$\leftarrow$ Decay density

$$\mathcal{M}^{\Lambda^*}_{\lambda_{\Lambda_b^0}, \lambda_p, \Delta\lambda_\mu} \equiv \sum_n \sum_{\lambda_{\Lambda^*}} \sum_{\lambda_\psi} \mathcal{H}^{\Lambda_b^0 \to \Lambda_n^* \psi}_{\lambda_{\Lambda^*}, \lambda_\psi} D^{\frac{1}{2}}_{\lambda_{\Lambda_b^0}, \lambda_{\Lambda^*} - \lambda_\psi}(0, \theta_{\Lambda_b^0}, 0)^* \mathcal{H}^{\Lambda_n^* \to Kp}_{\lambda_p, 0} D^{J_{\Lambda_n^*}}_{\lambda_{\Lambda^*}, \lambda_p}(\phi_K, \theta_{\Lambda^*}, 0)^* R_{\Lambda_n^*}(m_{Kp})\, D^1_{\lambda_\psi, \Delta\lambda_\mu}(\phi_\mu, \theta_\psi, 0)^*,$$

$\leftarrow$ Amplitudes for intermediate resonances

$$\mathcal{M}^{P_c}_{\lambda_{\Lambda_b^0}, \lambda_p^{P_c}, \Delta\lambda_\mu^{P_c}} \equiv \sum_j \sum_{\lambda_{P_c}} \sum_{\lambda_\psi^{P_c}} \mathcal{H}^{\Lambda_b^0 \to P_{cj} K}_{\lambda_{P_c}, 0} D^{\frac{1}{2}}_{\lambda_{\Lambda_b^0}, \lambda_{P_c}}(\phi_{P_c}, \theta^{P_c}_{\Lambda_b^0}, 0)^* \mathcal{H}^{P_{cj} \to \psi p}_{\lambda_\psi^{P_c}, \lambda_p^{P_c}} D^{J_{P_{cj}}}_{\lambda_{P_c}, \lambda_\psi^{P_c} - \lambda_p^{P_c}}(\phi_\psi, \theta_{P_c}, 0)^* R_{P_{cj}}(m_{\psi p})\;\; D^1_{\lambda_\psi^{P_c}, \Delta\lambda_\mu^{P_c}}(\phi_\mu^{P_c}, \theta_\psi^{P_c}, 0)^*, \quad (4)$$

$$\mathcal{H}^{A \to BC}_{\lambda_B, \lambda_C} = \sum_L \sum_S \sqrt{\tfrac{2L+1}{2J_A+1}} B_{L,S} \left(\begin{array}{cc|c} J_B & J_C & S \\ \lambda_B & -\lambda_C & \lambda_B - \lambda_C \end{array} \right) \times \left(\begin{array}{cc|c} L & S & J_A \\ 0 & \lambda_B - \lambda_C & \lambda_B - \lambda_C \end{array} \right),$$

$\leftarrow$ Complex couplings

$$R_X(m) = B'_{L^X_{\Lambda_b^0}}(p, p_0, d) \left(\frac{p}{M_{\Lambda_b^0}} \right)^{L^X_{\Lambda_b^0}} \mathrm{BW}(m|M_{0X}, \Gamma_{0X})\, B'_{L_X}(q, q_0, d) \left(\frac{q}{M_{0X}} \right)^{L_X}.$$

$\leftarrow$ Dynamical term

$$\mathrm{BW}(m|M_{0X}, \Gamma_{0X}) = \frac{1}{{M_{0X}}^2 - m^2 - iM_{0X}\Gamma(m)},$$

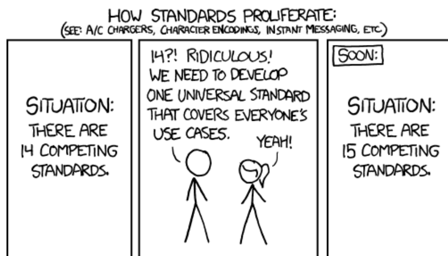
$$\Gamma(m) = \Gamma_{0X} \left(\frac{q}{q_0} \right)^{2L_X+1} \frac{M_{0X}}{m} B'_{L_X}(q, q_0, d)^2,$$

Alternative approach: formalism with covariant tensors (more expensive computationally).

Amplitude analysis tools in LHCb

Writing an amplitude fitting code from scratch is painful and time consuming. Several frameworks are in use at LHCb:

- **Laura++**
  - A powerful tool for traditional 2D Dalitz plot analyses (including time-dependent)
  - Single-threaded, but many clever optimisations
- **MINT**
  - Can do 3-body as well as 4-body final states
- **GooFit**
  - GPU-based fitter, able to do amplitude fits.
- **Ipanema-β**
  - GPU-based, `python` interface (`pyCUDA`)
- **qft++**
  - Not a fitter itself, but a tool to operate with covariant tensors



... and a lot of private code in use.

Amplitude analysis tools in LHCb

The problem with existing frameworks is that they are not very hackable. Trying to do something not foreseen in the framework design becomes a pain.

- Non-scalars in the initial/final states
- Complicated relations between fit parameters
- Fitting projections of the full phase space/partially-rec decays

For the analyses that go beyond a simple 2D amplitude, need a more flexible solution

- Still efficient from the computational point of view
- But not forget the tradeoff between person$\times$hours to implement the code vs. CPU$\times$hours to do the actual fits.

... and this is where AI comes to help

Machine learning tools for amplitude analyses?

Amplitude analyses

- Large amounts of data
- Complex models
- ... which depend on optimisable parameters
- Optimise by minimising neg. log. likelihood (NLL)
- Need tools which allow
  - Convenient description of models
  - Efficient computations

Machine learning

- Large amounts of data
- Complex models
- ... which depend on optimisable parameters
- Optimise by minimising cost function
- Need tools which allow
  - Convenient description of models
  - Efficient computations

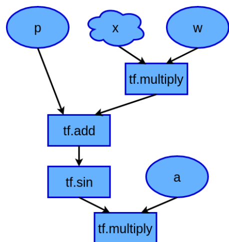
Many software tools are developed for machine learning, could reuse some of them in HEP analyses.

TensorFlow library

[Tensorflow webpage]
[White Paper]

- “TensorFlow is an open source software library for numerical computation using data flow graphs.” Released by Google in October 2015.
- Uses **declarative programming** paradigm: instead of actually running calculations, you describe what you want to calculate (*computational graph*)
- TF can then do various operations with your graph, such as:
  - Optimisation (e.g. caching data, *common subgraph elimination* to avoid calculating same thing many times).
  - Compilation for various architectures (multicore, multithreaded CPU, GPU, distributed clusters, mobile platforms).
  - Analytic derivatives to speed up gradient descent.
- Has `Python`, `C++` and `Java` front-ends. `Python` is more developed and (IMO) more convenient. Faster development cycle, more compact and readable code.

TensorFlow: basic structures



TF represents calculations in the form of directional *data flow graph*.

- Nodes: operations
- Edges: data flow

```
f = a*tf.sin(w*x + p)
```

Data are represented by *tensors* (arrays of arbitrary dimensionality)

- Most of TF operations are **vectorised**, e.g. `tf.sin(x)` will calculate element-wise $\sin x_i$ for each element x_i of multidimensional tensor `x`.
- Useful for ML fits, need to calculate same function for each point of large dataset.

Input structures are:

- *Placeholders*: abstract structure which is assigned a value only at execution time. Typically used to feed training data (ML) or data sample to fit to (our case).
- *Variables*: assigned an initial value, can change the value over time. Tunable parameters of the model.

TensorFlow: graph building and execution

To build a graph, you define inputs and TF operations acting on them:

```
import tensorflow as tf

# define input data (x) and model parameters (w,p,a)
x = tf.placeholder( tf.float32, shape = ( None ) )
w = tf.Variable( 1. )
p = tf.Variable( 0. )
a = tf.Variable( 1. )

# Build calculation graph
f = a*tf.sin(w*x + p)
```

(note that calculation graph is described using TF building blocks. Can't use existing libraries directly)

Nothing is executed at this stage. The actual calculation runs in the TF *session*:

```
# Create TF session and initialise variables
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Run calculation of y by feeding data to tensor x
f_data = sess.run( f, feed_dict = { x : [1., 2., 3., 4.] })

print y_data   # [ 0.84147096  0.90929741  0.14112    -0.7568025 ]
```

Input/output in `sess.run` is **numpy arrays**.

TensorFlow: minimisation algorithms

TensorFlow has its own minimisation algorithms:

```
# Placeholder for data
y = tf.placeholder( tf.float32, shape = ( None ) )

# Define chi2 graph using previously defined function f
chi2 = (f-y)**2

# TF optimiser is a graph operation as well
train = tf.train.GradientDescentOptimizer(0.01).minimize( chi2 )

# Run 1000 steps of gradient descent inside TF session
for i in range(1000) :
  sess.run(train, feed_dict = {
           x : [1., 2., 3., 4., 5.],     # Feed data to fit to
           y : [3., 1., 5., 3., 2.] } )
  print sess.run( [a,w,p] )   # Watch how fit parameters evolve
```

- Built-in minimisation functions seem to be OK for ANN training, but not for physics (no uncertainties, likelihood scans)
- MINUIT seems more suitable. Use it instead, and run TF only for likelihood calculation (custom FCN in python, run Minuit using PyROOT).

TensorFlow: useful features for amplitude analyses

Analytic gradient

Extremely useful feature of TF is automatic calculation of the graph for analytic gradient of any function (speed up convergence!)

```
tfpars = tf.trainable_variables()  # Get all TF variables
grad = tf.gradients(chi2, tfpars)  # Graph for analytic gradient
```

This is called internally in the built-in optimizers, but can be called explicitly and passed to `MINUIT`.

Partial execution

In theory, TF should be able to identify which parts of the graph need to be recalculated (after, e.g. changing value of `tf.Variable`), and which can be taken from cache.
In practice, this does not always work as expected, but there is a possibility to *inject* a value of a tensor in `sess.run` using `feed_dict` manually.

Interface with sympy

sympy is a symbolic algebra system for python. Consider it as mathematica with python interface. Free and open-source.

sympy has many extensions for physics calculations
See. e.g. sympy.physics module.

Recent versions of sympy can *generate code* for TensorFlow. Avoid re-implementing functions missing in TF. E.g. create TF tree for Wigner d function:

```
def Wignerd(theta, j, m1, m2) :
  """
  Calculate Wigner small-d function. Needs sympy.
    theta : angle
    j : spin
    m1 and m2 : spin projections
  """
  from sympy.abc import x
  from sympy.utilities.lambdify import lambdify
  from sympy.physics.quantum.spin import Rotation as Wigner
  d = Wigner.d(j, m1, m2, x).doit().evalf()
  return lambdify(x, d, "tensorflow")(theta)
```

TensorFlowAnalysis package

Project in gitlab: [TensorFlowAnalysis].

TF can serve as a framework for maximum likelihood fits (and amplitude fits in particular). Missing features that need to be added:

- ROOT interface to read/write ntuples (or use root-numpy)
- MINUIT interface for minimisation.
- Library of HEP-related functions.

Simplified standalone Dalitz plot generation/fitting script using only TF and ROOT. [DemoDalitzFit.py]

Only around 200 lines of Python, thanks to very compact code, e.g.:

```
def RelativisticBreitWigner(m2, mres, wres) :
  return 1./Complex(mres**2-m2, -mres*wres)

def UnbinnedLogLikelihood(pdf, data_sample, integ_sample) :
  norm = tf.reduce_sum(pdf(integ_sample))
  return -tf.reduce_sum(tf.log(pdf(data_sample)/norm ))
```

TensorFlowAnalysis: structure

Unlike many other amplitude analysis frameworks, `TensorFlowAnalysis` is basically a **collection of standalone functions** for components of the amplitude. These are then glued together in TF itself.
Components of the library are:

- Phase space classes (Dalitz plot, four-body, baryonic 3-body, angular etc.): provide functions to check if variable is inside the phase space, to generate uniform distributions etc.
- Fit parameter class: derived from `tf.Variable`, adds range, step size etc. for `MINUIT`
- Interface for `MINUIT`, integration, unbinned log. likelihood
- Functions for toy MC generation, calculation of fit fractions.
- Collection of functions for amplitude description:
  - Lorentz vectors: boosting, rotation
  - Kinematics: two-body breakup momentum, helicity angles
  - Helicity amplitudes, Zemach tensors
  - Dynamics: Breit-Wigner functions, form factors, non-resonant shapes
  - Elements of covariant formalism (polarisation vectors, γ matrices, etc.)
  - Multilinear interpolation of ROOT histograms

TensorFlowAnalysis: structure of a fitting script

Experimental data are represented in `TensorFlowAnalysis` as a 2D tensor

```
data[candidate][variable]
```

where inner index corresponds to event/candidate, outer to the phase space variable. E.g. 10000 Dalitz plot points would be represented by a tensor of shape `(10000, 2)`.

In the fitting script, you would start from the definitions of phase space, fit variables and fit model:

```
phsp = DalitzPhaseSpace(ma, mb, mc, md) # Phase space

# Fit parameters
mass = Const(0.770)
width = FitParameter("width", 0.150, 0.1, 0.2, 0.001)
a = Complex( FitParameter("Re(A)", ...), FitParameter("Im(A)", ...) )

def model(x) :          # Fit model as a function of 2D tensor of data
  m2ab = phsp.M2ab(x) # Phase space class provides access to individual
  m2bc = phsp.M2bc(x) #    kinematic variables
  ampl = a*BreitWigner(mass, width, ...)*Zemach(...) + ...
  return Abs(ampl)**2
```

TensorFlowAnalysis: structure of a fitting script

Fit model $f(x)$ enters likelihood via data and normalisation terms:

$$-\ln \mathcal{L} = -\left(\sum \ln f(x_{\mathrm{data}}) - N_{\mathrm{data}} \ln \sum f(x_{\mathrm{norm}}) \right)$$

Create two graphs for the model as a function of data and normalisation sample placeholders:

```
model_data = model( phsp.data_placeholder )
model_norm = model( phsp.norm_placeholder )
```

Now can create normalisation sample, and read data e.g.

```
norm_sample = sess.run( phsp.RectangularGridSample(500,500) )
data_sample = ReadNTuple(...)
```

Create the graph for negative log. likehood:

```
norm = Integral( model_norm )
nll = UnbinnedNLL( model_data, norm )
```

And finally call `MINUIT` feeding the actual data and norm samples to placeholders

```
result = RunMinuit(sess, nll, { phsp.data_placeholder : data_sample ,
                                phsp.norm_placeholder : norm_sample } )
```

TensorFlowAnalysis: structure of a fitting script

Call to

```
result = RunMinuit(sess, nll, ... )
```

internally includes calculation of analytic gradient for NLL. See benchmarks below to get the idea how that helps.

Analyst has full control over how likelihood is constructed and what variables serve as free parameters.

Since NLL graph is defined separately, it is easy to construct custom NLLs for e.g. combined CPV-allowed fits of two Dalitz plots.

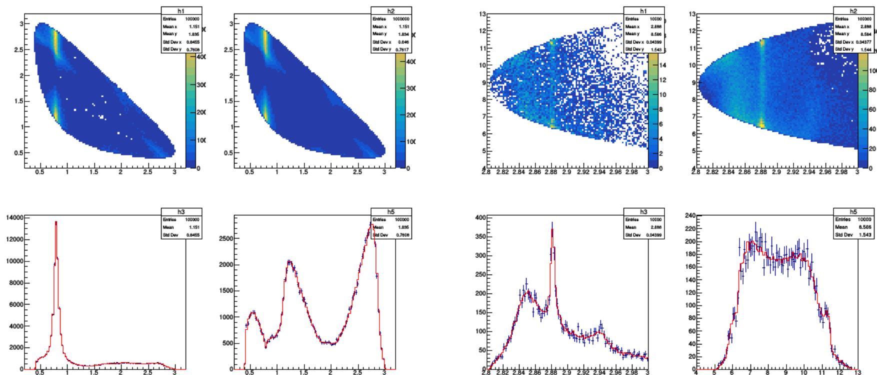
```
norm = Integral(model1_norm) + Integral(model2_norm)
nll = UnbinnedNLL(model1_data, norm) + UnbinnedNLL(model2_data, norm)
```

Similarly, complex combinations of fit parameters are easily constructed, e.g. CP-violating amplitudes

$$a_{\pm} = (\rho_{CPC} \pm \rho_{CPV}) e^{i(\delta_{CPC} \pm \delta_{CPV})}$$

Example: [$\Xi_b^- \to pK^-K^-$ CPV-enabled toy MC]

Isobar models implemented with helicity formalism and "simple" line shapes (Breit-Wigner, Gounaris-Sakurai, Flatté, LASS, Dabba, etc.).
Examples in [TensorFlowAnalysis/work]



Traditional Dalitz plot
$D^0 \to K_S^0 \pi^+ \pi^-$

Baryonic
$\Lambda_b^0 \to D^0 p \pi^-$

TensorFlowAnalysis: plans

Possible directions of development

- Extending library of functions: as needed by the analyses.
  - K-Matrix
  - New analytical couple-channel approaches
- Saving/loading of compiled graphs.
- Optimisations of CPU/memory usage, more intelligent caching.
- More use of symbolic maths.
- Self-documenting feature. Could use `Python` magic to automatically generate LaTeX description of formulas entering the fit (by replacing the input tensors with special `Python` objects).
- Automatic code generation: share stand-alone models with theorists.

CPU profiling

Profiling feature allows to identify bottlenecks in execution speed.

Breakup of operations by CPU core (32-core Xeon).

TensorFlowAnalysis: benchmarks

Benchmark runs (fit time only), compare 2 machines.
CPU1: Intel Core i5-3570 (4 cores @ 3.4GHz, 16Gb RAM)
GPU1: NVidia GeForce 750Ti (640 CUDA cores @ 1020MHz, 2Gb VRAM, 88Gb/s, 40 Gflops DP)
CPU2: Intel Xeon E5-2620 (32 cores @ 2.1GHz, 64Gb RAM)
GPU2: NVidia Quadro p5000 (2560 cores @ 1600MHz, 16Gb VRAM, 320Gb/s BW, 280 Gflops DP)
GPU3: NVidia K20X (2688 cores @ 732MHz, 6Gb VRAM, 250Gb/s BW, 1300 Gflops DP)

		Time, sec				
	Iterations	CPU1	GPU1	CPU2	GPU2	GPU3
$D^0 \to K_S^0\pi^+\pi^-$, 100k events, 500×500 norm.						
Numerical grad.	2731	488	250	113	59	82
Analytic grad.	297	68	36	18	12	19
$D^0 \to K_S^0\pi^+\pi^-$, 1M events, 1000×1000 norm.						
Numerical grad.	2571	3393	1351	937	306	378
Analytic grad.	1149	1587	633	440	148	180
$\Lambda_b^0 \to D^0 p\pi^-$, 10k events, 400×400 norm.						
Numerical grad.	9283	434	280	162	157	278
Analytic grad.	425	33	23	18	21	32
$\Lambda_b^0 \to D^0 p\pi^-$, 100k events, 800×800 norm.						
Numerical grad.	6179	910	632	435	266	364
Analytic grad.	390	133	62	126	32	45

$D^0 \to K_S^0\pi^+\pi^-$ amplitude: isobar model, 18 resonances, 36 free parameters
$\Lambda_b^0 \to D^0 p\pi^-$ amplitude: 3 resonances, 4 nonres amplitudes, 28 free parameters

TensorFlowAnalysis: problems and limitations

- At CERN, need to run on `lxplus7`/CENTOS.
- Memory usage: can easily exceed a few Gb of RAM for large datasets (charm) or complicated models.
  - Especially with analytic gradient
  - Limiting factor with consumer-level GPU.
- Double precision performance is essential
  - Single precision not sufficient, poor convergence.
- Performance issues with high-end GPUs
  - Slow RAM–VRAM data transfer for large datasets.
  - K20: only $\sim$ 50% GPU utilisation. VRAM latency or bandiwdth could be a bottleneck.
  - Try JIT (just-in-time) compilation? Can do kernel fusion and reduce RAM bandwidth requirement.
- Results are not 100% reproducible between different GPUs and CPU
  - Differences in FP implementation
- Less efficient than dedicated code developed with CUDA/Thrust, but *way* more flexible and easy to hack.

Examples in `TensorFlowAnalysis/work`

List of example fitting/toy MC scripts in the `master` branch of `TensorFlowAnalysis`

`AngularFit.py` Fit in 3D angular phase space a la $B^0 \to K^* \mu^+ \mu^-$

`D2KsPiPi.py` Realistic amplitude for $D^0 \to K^0_S \pi^+ \pi^-$ with 18 resonances, incl. background

`DalitzTF.py` Simplified amplitude for $D^0 \to K^0_S \pi^+ \pi^-$

`FourBodyToys.py` Toy MC generation for 4-body $\Lambda^0_b \to p\pi^- \pi^- \pi^+$

`HistInterpolation.py` Example of using interpolated 2D shape from ROOT histogram (e.g. for efficiency or background)

`Lb2Dppi.py` $\Lambda^0_b \to D^0 p \pi^-$ amplitude fit in helicity formalism

`Lb2DppiCovariantFit.py` $\Lambda^0_b \to D^0 p \pi^-$ amplitude fit in convariant formalism

`Lb2DppiCovariantToys.py` Toy MC generation of resonances in $\Lambda^0_b \to D^0 p \pi^-$ using convariant formalism

`Lc2pKpi.py` Realistic $\Lambda^+_c \to pK^- \pi^+$ amplitude using helicity formalism. Includes non-uniform efficiency

`Xib2pKK_CP.py` CPV-allowed combined fit of two Dalitz plots of $\Xi^-_b \to pK^- K^-$

Summary

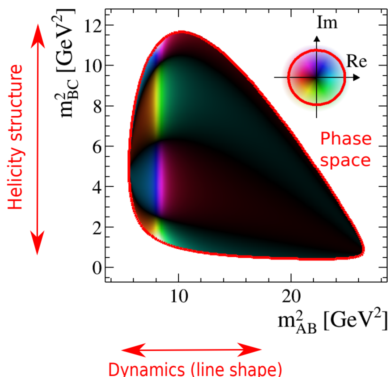
- Google made a good job providing us a functional framework for doing complicated fits: `TensorFlow`
- Why I think this approach is promising:
  - Can utilise modern computing architectures (mutithreaded, massively-parallel, distributed) without deep knowledge of their structure.
  - Interesting optimisation options, e.g. analytic derivatives help a lot for fits to converge faster.
  - Transparent structure of code. Only essence of things, no auxiliary low-level technical stuff in the description of functions.
  - Resulting models very portable and (with minor effort) can work standalone w/o the framework. Should be easy to e.g. share with theorists.
  - Flexible python interface can allow further tricks, e.g. automatic generation of LaTeX documentation or custom code generation.
  - Useful training value for students who will leave HEP for industry.
- As any generic solution, possibly not as optimal as specially designed tool. But taking development cycle into account, very competitive.
- `TensorFlowAnalysis` package: collection of functions to perform amplitude analysis fits. In active development, used for a few ongoing baryonic decay analyses at LHCb.

BACKUP

Introduction: amplitude analyses

Powerful tool to study complex decay dynamics, hadron spectroscopy, CP violation etc.
Perform fits of the multibody decay amplitude as a function of phase space variables

- Three-body decays $D \to ABC$: two kinematic variables M^2_{AB}, M^2_{BC} (Dalitz plot)
- Add angular variables if initial/final state not scalar
- More dimensions for > 3-body final states

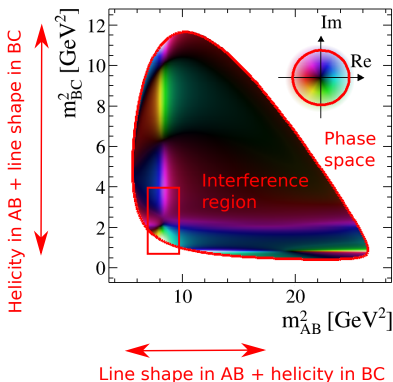


- Absolute phase not visible, but phases of components can be accessed though interference with other structures.
- Model-dependent fits: typically isobar model (sum of resonant/nonresonant components)
- More complicated models based on analyticity and unitarity.
- (Semi)-model-independent approaches: describe some partial waves as complex bins/splines, determine amplitude and phase through interference with other components.

Introduction: amplitude analyses

Powerful tool to study complex decay dynamics, hadron spectroscopy, CP violation etc.
Perform fits of the multibody decay amplitude as a function of phase space variables

- Three-body decays $D \to ABC$: two kinematic variables M^2_{AB}, M^2_{BC} (Dalitz plot)
- Add angular variables if initial/final state not scalar
- More dimensions for > 3-body final states



- Absolute phase not visible, but phases of components can be accessed though interference with other structures.
- Model-dependent fits: typically isobar model (sum of resonant/nonresonant components)
- More complicated models based on analyticity and unitarity.
- (Semi)-model-independent approaches: describe some partial waves as complex bins/splines, determine amplitude and phase through interference with other components.