

TensorFlow based fitters for amplitude analysis

Abhijit Mathad, on behalf of the developers

International Workshop on PWA and ATHOS, Bristol, UK

10th September 2021

University of Zurich UZH

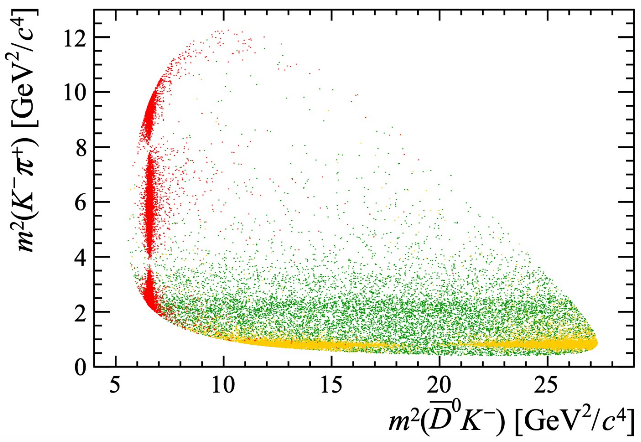
Amplitude or angular analysis

- **(Preaching the choir!)** Amplitude analysis is an important tool in studies such as hadron spectroscopy, finding exotic states, CP violation, effects of BSM, etc.
- **Where does TensorFlow enter? And what is it?**

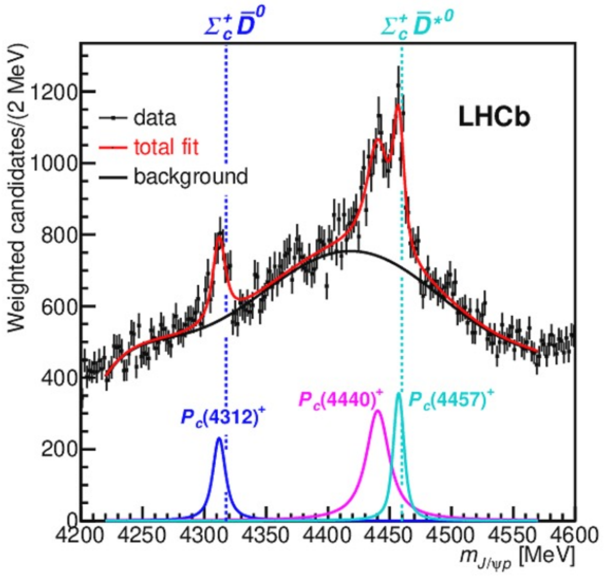
JCPC 2018 04 017

Phys. Rev. Lett. 122.222001

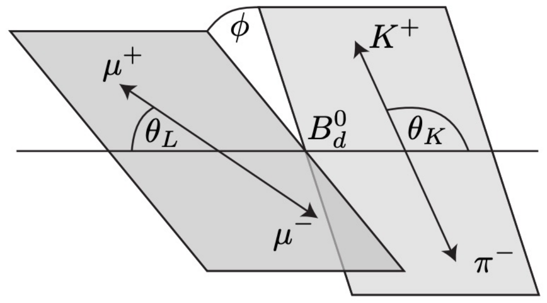
JHEP02(2016)104



Spectroscopy study and CPV: Toy Dalitz plot of $B_s^0 \rightarrow \overline{D}^0 K^- \pi^+$



Pentaquark discovery in $\Lambda_b \rightarrow J/\psi p K^-$



Constrain BSM effects: Angular analysis of rare and semi-leptonic decays

What is TensorFlow?

In what follows I will be only talking about TF v2.x, which is very different to TF v1.x!

TensorFlow

From Wikipedia, the free encyclopedia

TensorFlow is a free and open-source software library for machine learning. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.[4][5]

Tensorflow is a symbolic math library based on dataflow and differentiable programming. It is used for both research and production at Google.[6][7][8]

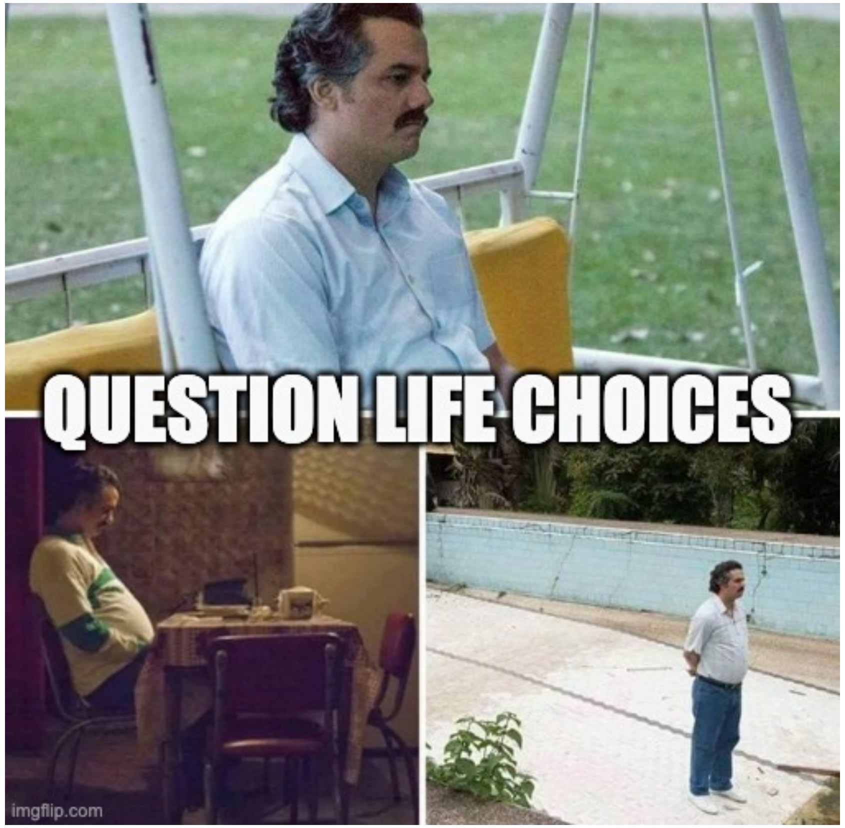
TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 in 2015.[1][9]

Machine learning (ML)? In a very general sense, the tools involved in doing an ML and Amplitude analysis are similar…

Path of an amplitude analysis

Test an idea with quick feasibility studies

It doesn't work!



Path of an amplitude analysis

Test an idea with quick feasibility studies

It works!

Collect large amounts of data and prune it

Extablish a model including all experimental effects

Obtain the parameters of interest

Evaluate associated uncertainties

Publish

Amplitude Vs Machine learning (ML) analysis

Test an idea with quick feasibility studies

↓

Collect large amounts of data and prune it

↓

Extablish a model including all experimental effects

↓

Obtain the parameters of interest

↓

Evaluate associated uncertainties

↓

Publish

Come up with an idea

↓

Collect data and identify useful features

↓

Build a ML model

↓

Train model and make predictions of target

↓

(Sometimes) Evaluate uncertainties

↓

Publish

There is quite some parallel between the two, so can we reuse the tools developed by a much broader ML community for our needs?

Welcome to the world of ML software!

There are lots of Machine learning software's freely available:

- Scikit-learn
- PyTorch
- TensorFlow
- Keras
- Weka
- KNIME
- The list goes on…

Covers most of our basic needs:

- Written in C++, python and CUDA.
- API's available for several languages (see here), with python being the main one.
- A plethora of mathematical operations and functions for numerical analysis (Very numpy-like!)
- Flexibility in developing a model with compact and readable code.
- Clever optimisations of code.
- Can run on various heterogeneous computing architectures (multi-core CPUs, TPU, GPU, CPU/GPU farm).

Hang on, we have our own packages in HEP? What are the advantages with TF?

Welcome to the world of HEP fitting frameworks!

Within LHCb several frameworks are used for amplitude/angular analysis:

- Laura++: C++ with dependency on ROOT, used in Dalitz plot analysis (including time-dependent), single threaded with many optimizations.
- Hammer: C++ interface, single threaded, mainly for semi-leptonic decays with missing neutrinos, has interface to RooFit (RooHammer).
- MINT : C++ interface to study generic 3-body and 4-body final states, has interface with LHCb simulation package Gauss.
- GooFit : GPU-based, C++ with python bindings.
- AmpGen: GPU-based, Amplitude analysis extension of GooFit.
- Ipanema-β: GPU-based with python interface (pyCUDA)
- qft++: Amplitude models in covariant formalism (no fitting yet).
- RooFit: Based on ROOT.
- CompPWA: C++ interface, also python (pycompwa), see dedicated talk.
- The list could go on…

Issues with the HEP fitting frameworks!

- They **lack functionality and/or flexibility** to cover all cases that might be encountered in an amplitude analysis.
- Significant alteration to the framework might be needed to accommodate outlying cases, e.g:
  - Non-scalars in the initial/final states.
  - Accommodating studies of partially reconstructed decays.
- For analysis that go beyond the available framework, need:
  - Speed of computation.
  - Speed of development.
  - Flexibility in model construction and fitting.

TensorFlow provides a lot of flexibility with quick model development and without compramising too much on speed!

Tensor in TensorFlow

TensorFlow

From Wikipedia, the free encyclopedia

TensorFlow is a free and open-source software library for machine learning. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.[4][5]

Tensorflow is a symbolic math library based on dataflow and differentiable programming. It is used for both research and production at Google.[6][7][8]

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 in 2015.[1][9]

- The data in TF is represented as a multi-dimensional array with rows and columns (just like numpy array).
  - Rows: Number of events
  - Columns: Dimensions of your phase space (or observables to fit). **Easily scalable to multi-dimensions.**
- This bulk data can be **mapped** (e.g. probability at various points in phase space) or **reduced** (e.g. fit fractions integrated over phase space).

Flow in TensorFlow

TensorFlow

From Wikipedia, the free encyclopedia

TensorFlow is a free and open-source software library for machine learning. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.[4][5]

Tensorflow is a symbolic math library based on dataflow and differentiable programming. It is used for both research and production at Google.[6][7][8]

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 in 2015.[1][9]

TF is based on a dataflow paradigm where a program is modelled as directed flow of data between mathematical operations. As such it first builds a ***computational graph*** that allows for:

- **Evaluation of analytic gradients** (through *automatic differentiation)* used by gradient based optimisers (e.g. Minuit).
- **Clever optimisations** (e.g data caching, *common subgraph elimination* that avoids multiple computations of same object).

Ok great! But what TF based frameworks are on the market?

Welcome to the world of TensorFlow frameworks!

Packages where only TF is used a computational backend!

zfit
Scalable pythonic fitting

AmpliTF

TensorFlowAnalysis2

VegasFlow

TF-PWA

Packages that use different computational backends such as:

- Numpy
- TensorFlow
- PyTorch
- JAX

pyhf
differentiable
Likelihoods

TensorWaves

However, I have experience with only zfit, AmplitTF and TFA2 only!

TF fitters

TensorWaves

Amplitude analysis package, see the dedicated talk. Code from sympy converted into different computational backends [Webpage].

TF-PWA

Amplitude analysis package based on TensorFlow, see the dedicated talk. [Webpage]

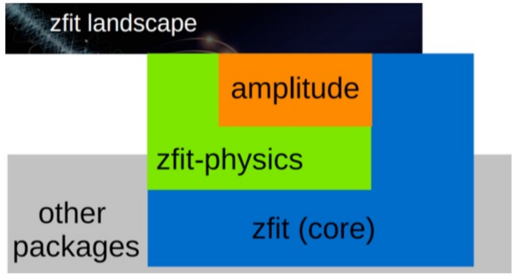
pyhf ∂ifferentiable ℒikelihoods

Not for amplitude analysis. Pyhf is a pure-python version of RooFit's HistFactory for template fitting! Also has different backends. Storing of likelihoods in human-readable format. [Webpage].

VegasFlow

Not a fitter but a MC integration library (with MC algorithms) on different computing architectures, compatible with python, C, C++ and Fortran [Webpage].

zFit

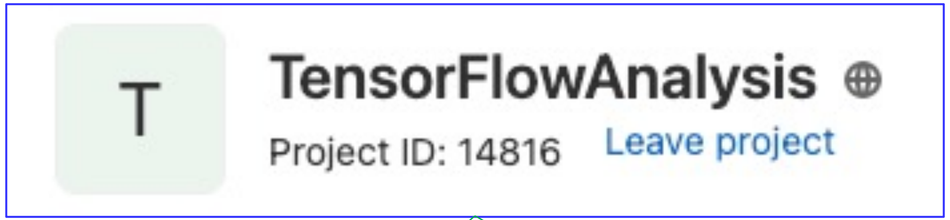


[Paper]
[Webpage]

- **Package focused on generic fitting (like RooFit). However very easy to implement custom amplitude model (see backup)!**
- Fully integrated with scikit-hep and can use hepstats for statistical inference ("RooStats-like") for sWeights, limits, etc.
- Can be easily interfaced with other amplitude analysis packages (like AmpliTF and TensorWaves). More on this later!
- Currently 2 LHCb papers published using this (both non-amplitude analysis) and a lot ongoing analyses (both within and outside of LHCb).
- A lot of functionality already present. In recent months, the focus has been on following aspects:
  - A lot of effort/thought has gone into providing support for a range of minimizers and homogenizing them (e.g. Ipyopt, NLopt, SciPy, TF).
  - Different integration techniques.
  - Binned fits (maybe in next release!).
  - Efficient convolution algorithms to include resolution information.

AmpliTF and TFA2

[TensorFlowAnalysis]



- Library based on TF v1.x
- I have conducted an LHCb analysis using this (Search for model-dependent CPV of $\Xi_b^- \to pK^-K^-$. See Anton's talk).
- Several ongoing LHCb analysis also use this library.

Update to TF v2.x and split into two separate packages

[AmpliTF]

AmpliTF

Collection of lot of useful functions for amplitude analysis! Easy to contribute and interface with other libraries!

[TFA2]

TensorFlowAnalysis2

Support for fitting, toy generation, easy plotting (LHCb publication style), multidimensional density estimation using neural nets, ROOT I/O with uproot.

Tasks in amplitude analysis

Many things to consider when doing amplitude analysis, such as…

Test an idea with quick feasibility studies	Construction of model and toy generation
↓	
Establish a model including all experimental effects	Parametrising efficiency, background and accounting for resolution.
↓	
Obtain the parameters of interest	Likelihood minimisation
↓	
Evaluate associated uncertainties	Generation and fitting to toys with small tweaks to the full model
↓	
Publish	Analysis reproducibility and preservation

How can AmpliTF and TFA2 help me each of these steps?

Model construction and toy generation

Test an idea with quick feasibility studies	**Construction of model and toy generation**
↓	
Establish a model including all experimental effects	Parametrising efficiency, background and accounting for resolution.
↓	
Obtain the parameters of interest	Likelihood minimisation
↓	
Evaluate associated uncertainties	**Generation and fitting to toys with small tweaks to the full model**
↓	
Publish	Analysis reproducibility and preservation

Lets look with model construction and toy generation with AmpliTF and TFA2...

Model construction: Phase space

[TFA2]
[AmpliTF]

The phase space model of of $\Lambda_c \to pK^-\pi^+$ can be constructed as follows:

```
from amplitf.phasespace import Baryonic3BodyPhaseSpace

MLc      = 2286.45992749e-3    #GeV
Mp       = 0.938
Mk       = 0.497
Mpi      = 0.140

phase_space = Baryonic3BodyPhaseSpace(Mp, Mk, Mpi, MLc)
uniform_phsp_sample = phase_space.uniform_sample(10000)
```

Phase space observables here include:

- $m^2_{pK}, m^2_{K\pi}, \cos(\theta_p), \phi_p$ and $\phi_{K\pi}$

Example of functionalities provided by AmplitTF in such `phasespace` objects:

- Ensure that a given phsp point is inside the phase space (Note: Non-rectangular phase space due to Dalitz plot observables).
- Generation of uniform samples in phase space or rectangular grid points in phase space (useful in multi-dimensional MC integration).
- In presence of identical particles (e.g. $\Xi_b^- \to pK^-K^-$) fold the conventional Dalitz plot (convDP).
- Calculation of various helicity angles and square Dalitz plot variables.
- In presence of narrow resonances (e.g. $\Lambda(1520)$) generate uniform sample in square Dalitz plot (sqDP) instead of conventional.
    - This is helpful in MC integration. There is also a method in this class that provides jacobian of sqDP -> convDP.
- Given the phsp points, returns the 4-vectors of final state in parent rest frame.

Model construction: Phase space

[TFA2]
[AmpliTF]

Different phase spaces are also available!

- baryonic_3body_phasespace.py — We just looked this 5D phase space!
- combined_phasespace.py — Combining multiple phase spaces helpful in sequential decays.
- dalitz_phasespace.py — 2D Dalitz phase space (forms base class for Baryonic3BodyPhaseSpace).
- four_body_angular_phasespace.py
- four_body_helicity_phasespace.py

5D and 3D phase space of 4-body decay.

- lambda_phasespace.py
- rectangular_phasespace.py — Mulit-dimensional rectangular phase space. Helpful for sequential decays.
- veto_phasespace.py

Accept/veto certain regions of phase space.

Model construction: Dynamic terms

[TFA2]
[AmpliTF]

Define the resonance properties

```
import amplitf.dynamics as atfd
import tfa.optimisation as tfo

resonances = {}
resonances["Kstar_kpi(892)"] = {
    "lineshape" : atfd.breit_wigner_lineshape,
    "mass"      : tfo.FitParameter("Kstar_kpi(892)_mass", 0.89, 0.,2., 0.01)  #tfo.FitParameter(name, init_val, min, max, step_size)
    "width"     : tfo.FitParameter("Kstar_kpi(892)_width",0.044,0.03,0.05, 0.01)
    "spin"      :  2,
    "parity"    : -1,
    "coupl"     : [ atfi.const(1.) ,
                    atfi.const(0.),
                    tfo.FitParameter("Kstar_kpi(892)_real_1", 5.0, -10., 10., 0.01),
                    tfo.FitParameter("Kstar_kpi(892)_imag_1", 5.0, -10., 10., 0.01),
                    tfo.FitParameter("Kstar_kpi(892)_real_2", 5.0, -10., 10., 0.01),
                    tfo.FitParameter("Kstar_kpi(892)_imag_2", 5.0, -10., 10., 0.01),
                    tfo.FitParameter("Kstar_kpi(892)_real_3", 5.0, -10., 10., 0.01),
                    tfo.FitParameter("Kstar_kpi(892)_imag_3", 5.0, -10., 10., 0.01),]
    "l_lamc"    : int( _l_bc_k( 2 , Jp , Jlam_c ) ),
    "l_res"     : 1
}
```

Helpful functions provided by AmpliTF.

Side note: The wigner-D function actually uses sympy (open source symbolic python library) to generate TF code.

$$A^{K^*}_{M_{\Lambda_c},\lambda_p} = C_4 \times R(m^2(K^-\pi^+))\times \sum_{\lambda_{K^*}} \left(D^{J_{\Lambda_c},*}_{M_{\Lambda_c},\lambda_{K^*}-\lambda_p}(\phi^{[\Lambda_c]}_{K^*},\theta^{[\Lambda_c]}_{K^*},0) \times D^{J_{K^*},*}_{\lambda_{K^*},0}(\phi^{[K^*]}_{K},\theta^{[K^*]}_{K},0) \times \epsilon_{\lambda_p} \times H^{J_{\Lambda_c},J_{K^*}}_{\lambda_{K^*},\lambda_p} \right)$$

Convert equations to code easily

```
def _Ampl_K( res, Nu , lp , lres, coupling ):
    A  = tf.math.conj( atfk.wigner_capital_d( kstar_phi_lc , kstar_theta_lc  , 0 , self.Jlam_c  , Nu , lres - lp ) )
    A *= tf.math.conj( atfk.wigner_capital_d( k_phi_kstar  , k_theta_kstar   , 0 , res_val[res]['spin'] , lres , 0 ) )
    A *= atfi.cast_complex( ( -1 )**( self.Jp/2. - lp/2. ) )
    A *= coupling
    A *= res_val[res]['lineshape']( m2_kpi , res_val[res]["mass"]() , res_val[res]["width"]() , self.Mk , \
                                   self.Mpi , self.Mp , self.MLc , d_res , d_Lambda_c , \
                                   res_val[res]['l_res'] , res_val[res]['l_lamc'])
    return A
```

Model construction: Dynamic terms

[TFA2] [AmpliTF]

Definitions of various lineshape related function: RBW, non-resonant polynomial, Flatte, LASS, Gounaris-Sakurai, etc

Calculation of integrals, weighted/un-weighted unbinned likelihood function.

- dalitz_decomposition.py
- dynamics.py
- interface.py
- kinematics.py
- likelihood.py

Functions for calculating angular observables from lorentz invariant quantities according to Dalitz Plot Decomposition (DPD) technique.

Functions that interface with TF, setting precision, etc.

4-vectors, rotations, boost, Wigner-D functions, calculation of various angles in 3 and 4 body decays, etc.

Toy generation

[TFA2]
[AmpliTF]

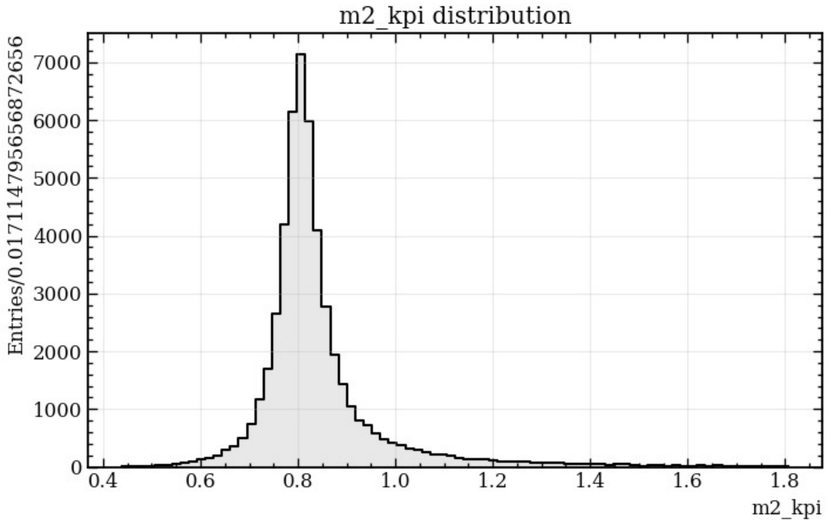
```
import tfa.toymc as tft
import amplitf.dynamics as atfd
import tfa.plotting as tfp

#define the model with amplitude
def model(x):
  return atfd.density(ampl)

#Define phase space
phase_space = Baryonic3BodyPhaseSpace(Mp, Mk, Mpi, MLc)
#get maxima for the pdf for toy generation
maximum     = tft.maximum_estimator(model, phase_space, size_of_sample_for_max) * 1.5
#accept/reject method for toy generation
toy_sample  = tft.run_toymc( model, phase_space, size , maximum, chunk)

#plot the 1d distribution in lhcb style
tfp.set_lhcb_style(size=12, usetex=False)
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(4, 3))
data    = toy_sample.numpy()
tfp.plot_distr1d(data, bins= 80, range=(np.min(data), np.max(data)), ax=ax, label = "m2_kpi")
```

Output



Experimental effects in the model

Test an idea with quick feasibility studies — Construction of model and toy generation

↓

Establish a model including all experimental effects — **Parametrising efficiency, background and accounting for resolution**.

↓

Obtain the parameters of interest — Likelihood minimisation

↓

Evaluate associated uncertainties — Generation and fitting to toys with small tweaks to the full model

↓

Publish — Analysis reproducibility and preservation

Lets look at parametrising efficiency and background with AmpliTF and TFA2…
(Note: No custom convolution functions yet provided by the library to account for resolution)

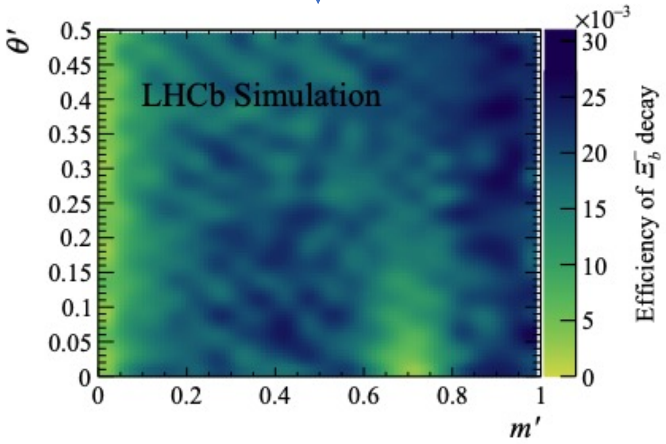
Interpolation of efficiency

[TFA2]
[AmpliTF]

Efficiency maps usually obtained from ratio of histograms before and after selections. To mitigate discontinuity at bin edges use smoothing techniques. AmpliTF provides a method (`interpolate`) for multi-linear interpolation.

```
#Read the ROOT TFile and the TH2 object (efficiency map as function of sqDP)
f = TFile(fname, "read")
h = f.Get("eff_sqDP")
#Convert TH2D object to RootHistShape (Note only in TFA and not TFA2)
eff_roothistshape = RootHistShape(h)
#Convert data (convDP points) into sqDP points
sqDP        = tf.stack([phase_space.MPrimeAC(convDP), phase_space.ThetaPrimeAC(convDP)],axis=1)
#Obtain the linearly interplated eff values at a given sqDP points
eff         = eff_roothistshape.shape(sqDP)
```

Output



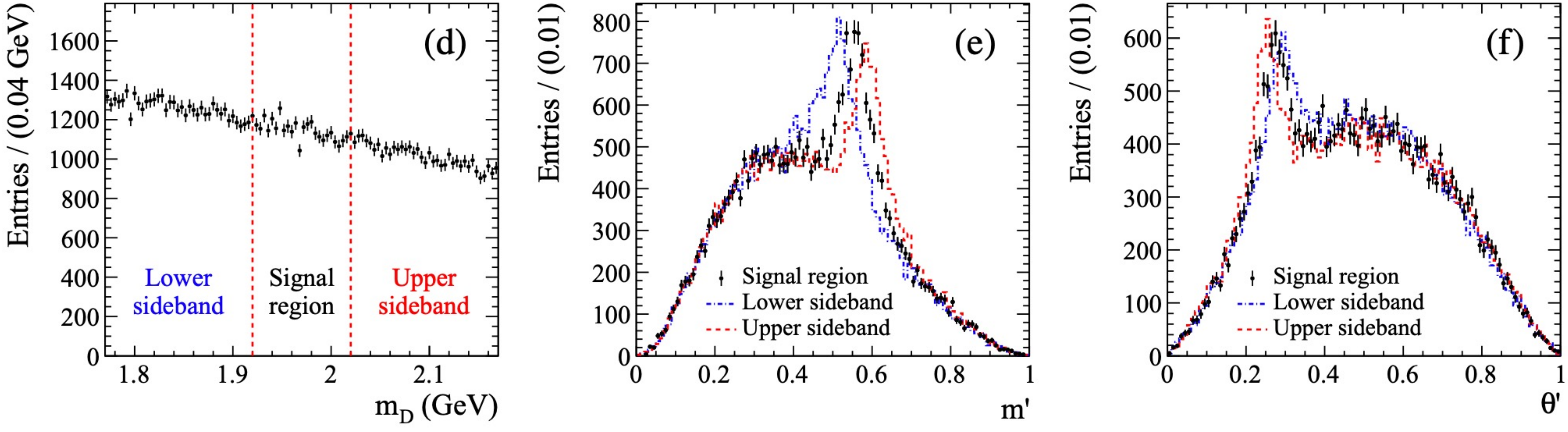
[LHCb-PAPER-2020-017]

Side note about code snippet: The `RootHistShape` class takes TH2D and calls the `interpolate` method internally. This class only exists in TFA and not TFA2 (since latter wanted to be ROOT independent).

Modelling of the combinatorial background [TFA2] [AmpliTF]

The combinatorial background is usually modelled using B mass sideband region and as a result the B mass constraint affects its distribution under the signal region.

Simulated sample of $D_s \to K\pi\pi$



JINST 16 P06016

Modelling of the combinatorial background [TFA2] [AmpliTF]

To avoid effects of B mass constraint, in the analysis of model-dependent CPV in $\Xi_b^- \to pK^-K^-$ [LHCb-PAPER-2020-017], artificial neural network (ANN) was trained to model the 3D distribution of sqDP and Ξ_b^- mass (see JINST 16 P06016, the talk and code here). This model is then used to extrapolate the sqDP distribution of the bkg at the Ξ_b^- mass.

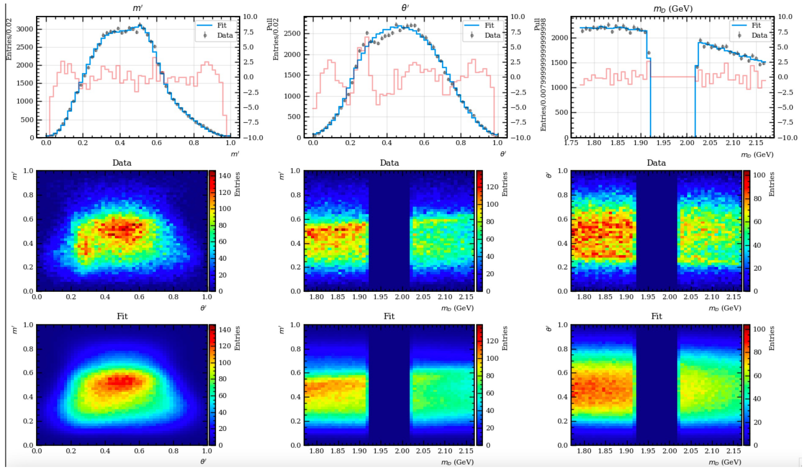
```
import tfa.neural_nets as tfn

#sqDP phase space
sqdlz_phsp = RectangularPhaseSpace( ((0., 1.), (0., 1.)) )
#m(B) phase space (Here considered Ds->Kpipi decay)
m_phsp     = RectangularPhaseSpace( ((1.97-0.2, 1.97+0.2), ) )
#Combine spDP and m(B)
observables_phase_space = CombinedPhaseSpace(sqdlz_phsp, m_phsp)
#Veto the signal region
exp_phase_space = VetoPhaseSpace(observables_phase_space, 2, (1.97-0.05, 1.97+0.05) )

#train the model
tfn.estimate_density(
    exp_phase_space, # Phase space
    data,            # Data to train with
    ranges=bounds,   # list of ranges for observables
    labels=titles,   # Ovservables' titles
    learning_rate=0.0002,  # Tunable meta-parameter
    n_hidden=[32, 64, 32, 8],  # Structure of hidden layers (4 layers, 64, 64, 32 and 8 neurons)
    training_epochs=50000,  # Number of training iterations (epochs)
    norm_size=4000000,    # Size of the normalisation sample
    print_step=50,        # Print cost function every 50 epochs
    display_step=500,     # Display status of training every 500 epochs
    initfile="init_3d.npy",  # Init file (e.g. if continuing previously interrupted training)
    outfile="train_3d",  # Name prefix for output files (.pdf, .npy, .txt)
    seed=4,  # Random seed
    fig=fig,  # matplotlib window references
    axes=ax)
```

Minimize the likelihood

Store the trained weights and biases

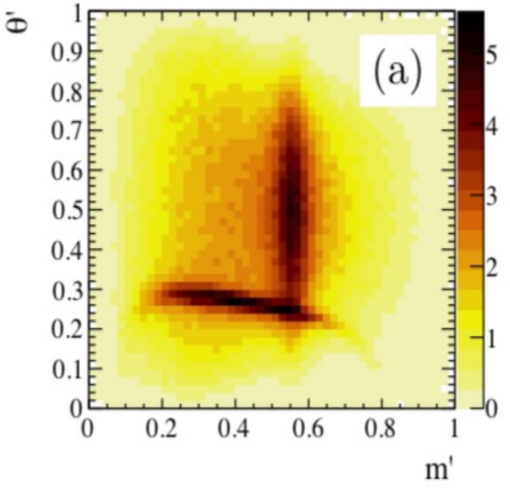


Side note: Takes a long time to train. Better run on GPU if available.

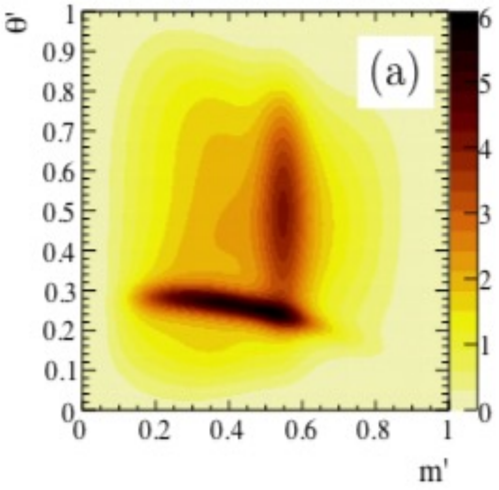
Modelling of the combinatorial background [TFA2] [AmpliTF]

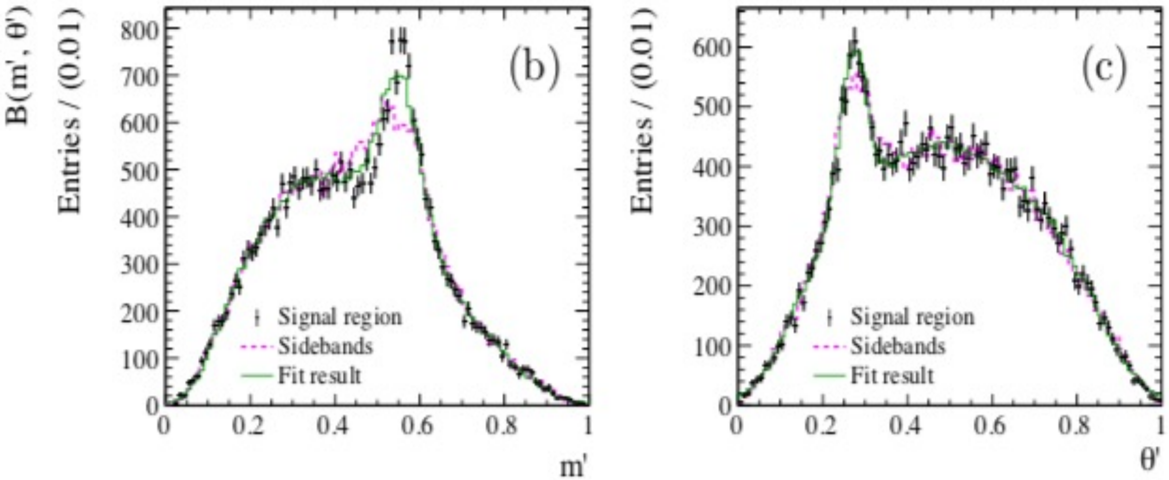
Load the trained weights and biases and generate the 3D sample to obtain the sqDP distribution in signal region

True distribution



Prediction





Side note: Situation can be improved by assissting ANNs and can also be applied to model efficiency.
[JINST 16 P06016, talk]

Minimisation

Test an idea with quick feasibility studies	Construction of model and toy generation
↓	
Establish a model including all experimental effects	Parametrising efficiency, background and accounting for resolution.
↓	
Obtain the parameters of interest	**Likelihood minimisation**
↓	
Evaluate associated uncertainties	Generation and fitting to toys with small tweaks to the full model
↓	
Publish	Analysis reproducibility and preservation

Lets look at likelihood minimisation with AmpliTF and TFA2…

Minimisation

[zfit] [TFA2] [AmpliTF]

- Define the function that calculates negative log likelihood function that takes list of `FitParameter` as input (see slide).
- Minimize using iminuit (since error estimates readily available).
- Note that one can also interface with zfit to use the range of minimizers available there.

```
minimizer = zfit.minimize.NLoptLBFGSV1()
#minimizer = zfit.minimize.Minuit()
#minimizer = zfit.minimize.ScipyTrusConstrV1()
#minimizer = zfit.minimize.IpyoptV1()
result     = minimizer.minimize(nll_func, parameter_list)
```

```
import tfa.optimisation as tfo
import amplitf.likelihood as atfl

#define NLL
def nll(data, norm):

    @atfi.function
    def _nll(pars):
        model      = fit_model(data, pars)
        model_norm = fit_model(norm, pars)
        intg       = atfl.integral(model_norm)
        return atfl.unbinned_nll(model, intg)

    return _nll

#conduct the minimisation
nll_func = nll(toy_sample, norm_sample)
results  = tfo.run_minuit(nll_func,
                pars, #List of FitParameters
                use_gradient=True,
                use_hesse = False,
                use_minos = False,
                get_covariance = False)
```

Speeding up the minimisation

[TFA2]
[AmpliTF]

- To establish an amplitude model, one first considers a large set of resonant and non-resonant components e.g., in $\Xi_b^- \to pK^-K^-$ analysis [LHCb-PAPER-2020-017] we initially started with 128 free parameters with data size of ~500 events a **single fitting taking 3 hours**.
- Can we improve? The decay density can be expressed as complex parameter-free function of phase space with factorizable and non-factorizable parameters of interest.

$$\frac{d\Gamma}{dm^2 d\Omega} = \sum h_i h_j^* \, * \, f_{ij}(\mathrm{m}^2 \,|\, m_0, \Gamma_0, \alpha) \, * \, g_{ij}(\Omega)$$

- In $\Xi_b^- \to pK^-K^-$ analysis, *since masses and widths were fixed*, we **pre-computed all the parameter free integrals** with and w/o efficiency information (Taylor expanding the exponential non-resonant lineshapes with a slope parameter). This gave massive speed gains i.e. **single fit now took 15 mins**!
- One can further **cache all the parameter free terms** too for data (see here).
- Some benchmark studies are mentioned in the backup, but with TFA.

Analysis reproducibility and preservation

Test an idea with quick feasibility studies	Construction of model and toy generation
↓	
Establish a model including all experimental effects	Parametrising efficiency, background and accounting for resolution.
↓	
Obtain the parameters of interest	Likelihood minimisation
↓	
Evaluate associated uncertainties	Generation and fitting to toys with small tweaks to the full model
↓	
Publish	**Analysis reproducibility and preservation**

The models written in TF are portable and can, with some effort, work standalone. This could perhaps be shared?

Issues and future

- Graph building impacts performance for large number of quick and simple fits.
- When large datasets are involved memory usage is high (> few Gb of RAM) even more when analytic gradient calculation is involved.
- Less efficient than code developed with CUDA, but very flexible!

- Support for covariant formalism (some code in TFA), K-matrix, etc.
- Porting of other useful code (like fit fraction and interference fit fractions) from TFA to AmpliTF.
- Develop some examples on how different fitting frameworks can benefit from each other.
- Documentation on AmpliTF and TFA2. Conventions, formalisms and formulae used!
- Evolve code in AmpliTF and TFA2, such that we are not locked into TF and can easily switch backends to numpy, numba, JAX, etc (like in TensorWaves).

Summary

- TensorFlow provides a lot of flexibility with quick model development and without compramising too much on speed!
- Code can be adapted easily to run on various computing architechures.
- Benefit a lot from inherent optimisations that come for free with TF and with optimisations implemented in amplitude analyses packages.
- Presented an overview of various high and low-level TF fitting frameworks.
- Highlighted here how AmpliTF and TFA2 can help in different steps of amplitude analysis [Demo scripts][Installation instructions and guide].

Backup

Implement a wave solutions in TF (Eager execution)

Data represented as multi-dimensional tensor object (`tf.Tensor`)

$$y(x) = a * \sin(w * x + p)$$

Parameters can be tunnable (`tf.Variable`) or non-tunnable (`tf.constant`).

```python
import tensorflow as tf

a = tf.constant(1.0)
w = tf.Variable(1.0, trainable=True)
p = tf.Variable(0.0, trainable=True)

def f(x):
    print('Data:', x)
    return a * tf.sin(w * x + p)

x = tf.constant([0.0, 1.0])
print('Function Value (TF tensor):', f(x))
print('Function Value (Numpy Array): ', f(x).numpy())
```

Outputs

```
Data: tf.Tensor([0. 1.], shape=(2,), dtype=float32)
Function Value (TF tensor): tf.Tensor([0. 0.84147096], shape=(2,), dtype=float32)
Data: tf.Tensor([0. 1.], shape=(2,), dtype=float32)
Function Value (Numpy Array):  [0.  0.84147096]
```

Note that the data is printed twice. Numpy-like behaviour or eager execution!

Lazy evaluation (@tf.function)

[Code: hello_world.py]

Same code as before but add a decorator (`@tf.function`) and execute.

Side note: For Just-in-time compilation (JIT): `@tf.function(jit_compile=True)`

```
import tensorflow as tf

a = tf.constant(1.0)
w = tf.Variable(1.0, trainable=True)
p = tf.Variable(0.0, trainable=True)

@tf.function
def f(x):
    print('Data:', x)
    return a * tf.sin(w * x + p)

x = tf.constant([0.0, 1.0])
print('Function Value (TF tensor):', f(x))
print('Function Value (Numpy Array): ', f(x).numpy())
```

Outputs

```
Data: Tensor("x:0", shape=(2,), dtype=float32)
Function Value (TF tensor): tf.Tensor([0. 0.84147096], shape=(2,), dtype=float32)
Function Value (Numpy Array):  [0.  0.84147096]
```

Note the data get printed only once and is different to before! What is hapenning?

Lazy execution (@tf.function)

- The decorator (`@tf.function`) takes the function (`f`) and returns a computational graph (with *nodes* as operations and *edges* representing acyclic flow of data).
- Under lazy evaluation this graph is compiled once, speeding up the code!

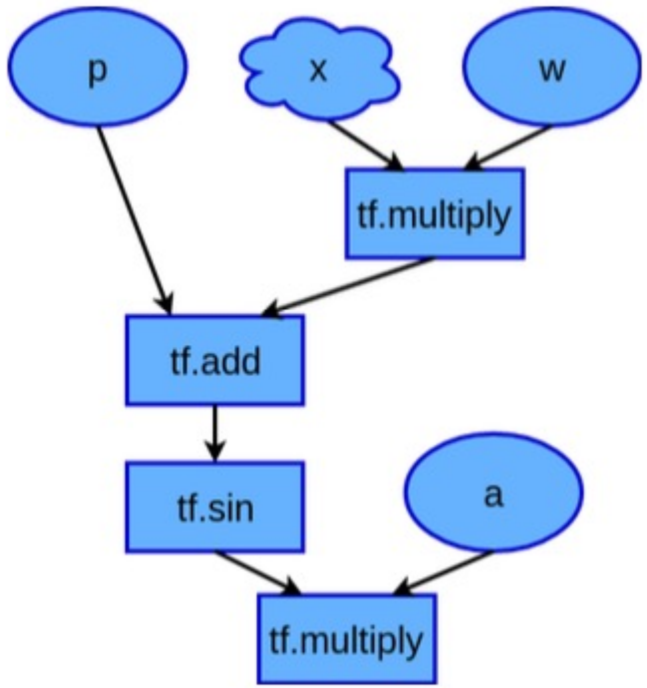
Side note: `tf.Variable.assign` can be used to write new value to the variable memory (w/o adding any new operations to the graph).

```python
import tensorflow as tf

a = tf.constant(1.0)
w = tf.Variable(1.0, trainable=True)
p = tf.Variable(0.0, trainable=True)

@tf.function
def f(x):
    print('Data:', x)
    return a * tf.sin(w * x + p)

x = tf.constant([0.0, 1.0])
print('Function Value (TF tensor):', f(x))
w.assign(0.1)
print('Function Value (TF tensor):', f(x))
```



Benefits of computational graph

- Many advantages, but a direct application is *auto differentiation*, i.e. applying chain rule as we traverse forwards/backwards through the graph, which helps in evaluation of analytic gradients.
- Such analytic gradients of the minimizing function not only help overcome the problems of numerical ways of computing gradients (round-off errors) but also help speed up the minimization itself!

```
#observed values
obs_vals = tf.constant([1.0, 2.0])

@tf.function
def chi2():
    exp_vals = f(x) #expected values
    return tf.reduce_sum((exp_vals - obs_vals) ** 2)

with tf.GradientTape() as gt:
    grad = gt.gradient(chi2(), [a, w, p])
print([g.numpy() for g in grad])
```

$\rightarrow$ Outputs

$\frac{\partial \chi^2}{\partial a}$ $\frac{\partial \chi^2}{\partial w}$ $\frac{\partial \chi^2}{\partial p}$

```
[-1.9497371, -1.2519118, -3.2519116]
```

zfit custom model

```
#define custom amplitude model
class AmplitudeModel(zfit.pdf.BasePDF):
    def __init__(self, name, params, obs):
        super().__init__(name=name, params=params, obs=obs)

    def _unnormalized_pdf(self):
        #write the model
        r = self.params['coupling_real']
        i = self.params['coupling_imag']
        #write your model in pure TF or AmpliTF
        pass

        #return an unnormalized pdf
        return

#define the space being fit
obs_space = zfit.Space(obs=['m2_kpi'], limits=limits_m2kpi)

#define parameter
parameters = {}
parameters['couplings_real'] = zfit.Parameter("couplings_real", 0., -10., 10., 0.01)
parameters['couplings_imag'] = zfit.Parameter("couplings_imag", 0., -10., 10., 0.01)

#define mode
model = AmplitudeModel("model", parameters, obs_space)

#get data
data = zfit.Data.from_root(root_file, root_tree, branches)

#define loss
my_loss = zfit.loss.UnbinnedNLL(model,data,constraints=constraint)

#minimize
minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(my_loss)
```

CPU profiling

Profiling feature allows to identify bottlenecks in execution speed.

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

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

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

Analysis reproducibility and preservation

- Analysts spend quite some time trying to reproduce results of a different analysis for various purposes (correcting simulation samples, extending the previous analysis, etc).
- To compare between different analysis papers usually publish fit fractions, interference fit fractions, etc instead of the direct fit results of the helicity couplings.
- There could also be intricacies in the model building that might be perhaps be overlooked in the paper.
- TensorFlow allows one to store computational graphs however this not human readable.
- **The models written in TF are portable and can, with some effort, work standalone. This could perhaps be shared? Should be cautious in propagating bugs!**