Partial wave analysis framework TF-PWA and related analysis

Yi Jiang （蒋艺）
University of Chinese Academy of Sciences (中国科学院大学)

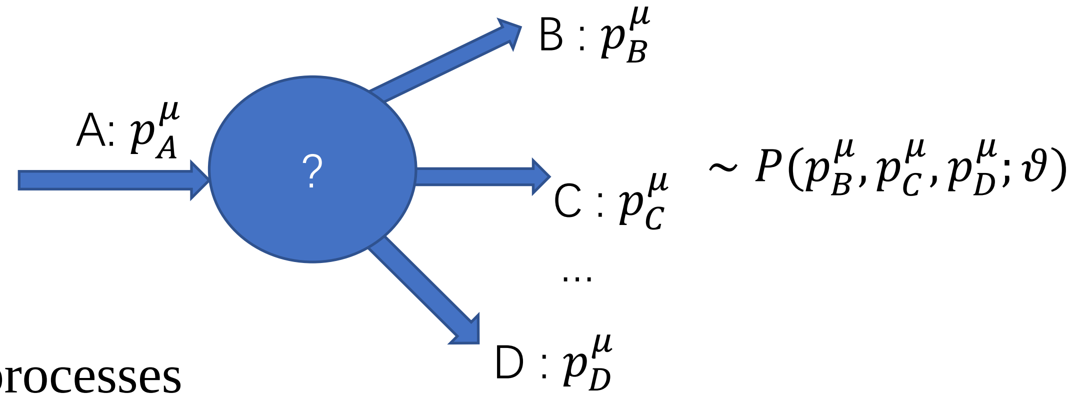
Jul 30, 2024, YanTai

Outline

- Introduction
- TF-PWA
  - Basic framework
  - Custom model in TF-PWA
  - Automatic Differentiation
  - Performances of TF-PWA
- Analysis using TF-PWA in LHCb
  - $B^+ \to D^{*\pm} D^{\mp} K^+$
- Summary

Introduction

- Amplitude analysis / Partial wave analysis (PWA) is a powerful method to study multi-body decay processes, e.g.
  - to search for (exotic) resonances and measure their properties
  - to understand CP violation over phase space



- Most of previous fitters are designed for special processes or are time-consuming.

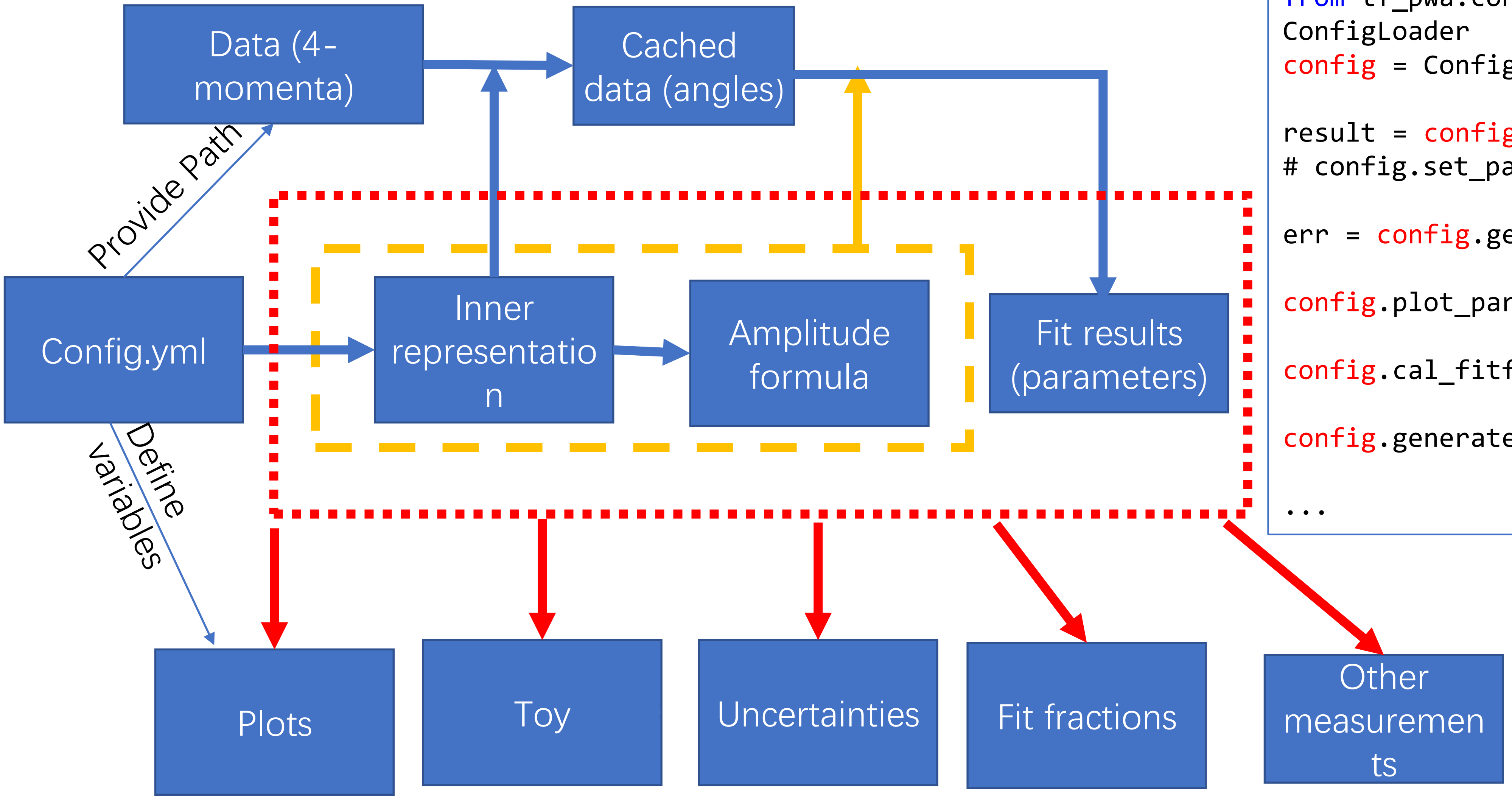
- A general PWA framework using modern acceleration technology (such as GPU, AD, …) is eagerly needed.

TF-PWA: Partial Wave Analysis with TensorFlow

- Fast
  - GPU based
  - Vectorized calculation
  - Automatic differentiation

    Quasi-Newton Method: scipy.optimize
- General
  - Custom model available
- Easy to use
  - Simple configuration file (example provided)
  - Most of the processing is automatic
  - All necessary functions implemented
  - Developing more functions
- Open access and well supported https://github.com/jiangyi15/tf-pwa

Configuration as global representation

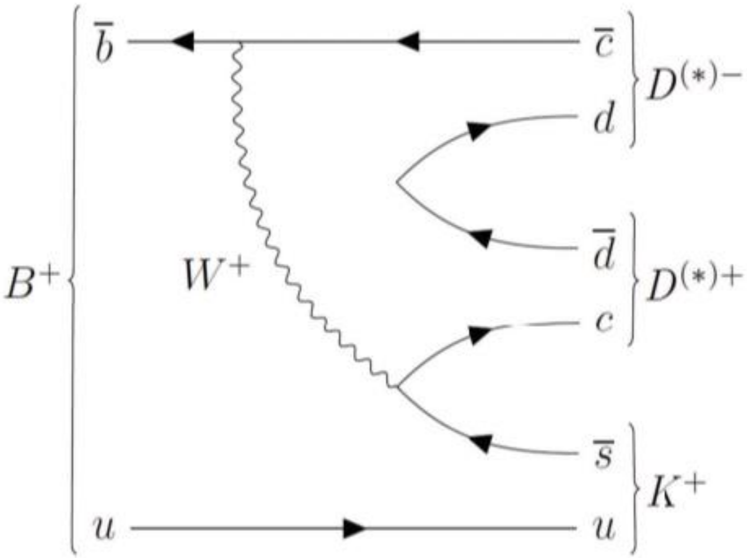



```python
from tf_pwa.config_loader import \
ConfigLoader
config = ConfigLoader("config.yml")

result = config.fit()
# config.set_params("final_params.json")

err = config.get_params_error()

config.plot_partial_wave()

config.cal_fitfractions()

config.generate_toy(10000)

...
```

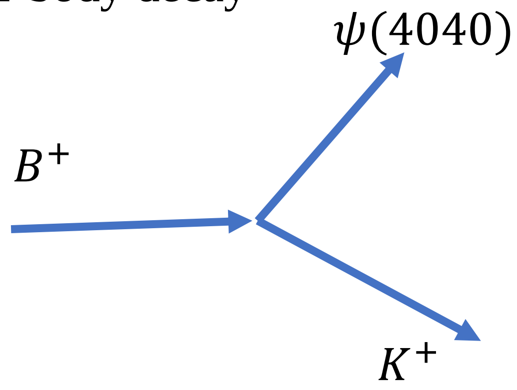
Configuration

YAML: https://yaml.org

- What is needed?
  - Particles (Resonances, line) and their properties
  - Decays (interaction, vertex) and their properties
- Store in dict or list, save as YAML file.
- Possible process in $B^+ \to \mathrm{D}^{*\pm}\mathrm{D}^{\mp}\mathrm{K}^{\pm}$



2-body decay



Serialize as

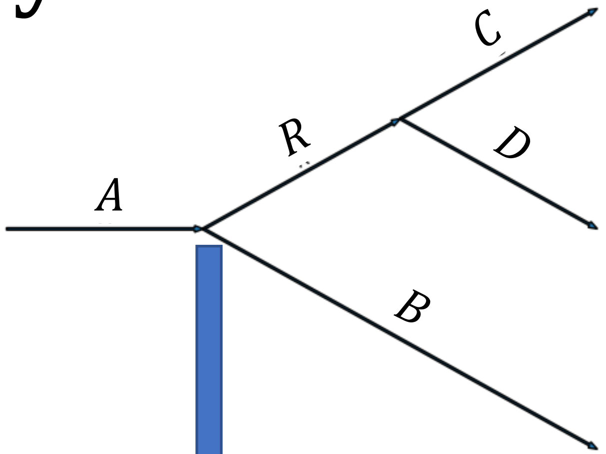
```
Bp: [
  [DstD, K],
  [DstK, D ],
  [DK, Dst ],
]
```

Config.yml In TF-PWA

```
psi(4040):
  J: 1
  P: -1
  mass:  4.040
  width:  0.080
  model: C(BWR)
…
```

Helicity formalism

•



$$A^R_{\lambda_A,\lambda'_B,\lambda'_C,\lambda'_D}$$

$$=$$

$$\sum_{\lambda} H_{\lambda_R \lambda_B} D^{j_A\star}_{\lambda_A,\lambda_R-\lambda_B}(\varphi_1,\theta_1,0) R(M) H_{\lambda_C,\lambda_D} D^{j_R\star}_{\lambda_R,\lambda_C-\lambda_D}(\varphi_2,\theta_2,0)$$

$$D^{j_B\star}_{\lambda_B,\lambda_B'}(\alpha_B,\beta_B,\gamma_B) D^{j_C\star}_{\lambda_C,\lambda_C'}(\alpha_C,\beta_C,\gamma_C) D^{j_D\star}_{\lambda_D,\lambda_D'}(\alpha_D,\beta_D,\gamma_D)$$

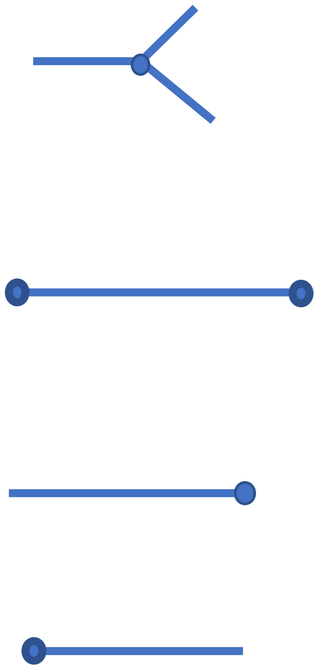
$$\frac{d\sigma}{d\Phi} \propto \sum_{\lambda_A} \sum_{\lambda_B,\lambda_C,\lambda_D} \left| \sum_R A^R_{\lambda_A,\lambda_B,\lambda_C,\lambda_D} \right|^2$$

Automatically calculated from decay structure

Feynman rules

Decay

Particle



User defined

$$A^{0\to 1+2} = H_{\lambda_1,\lambda_2} D^{j_0\star}_{\lambda_0,\lambda_1-\lambda_2}(\varphi,\theta,0)$$

Wigner-D matrix

$$R(M) = \frac{1}{m_0^2 - M^2 - i m_0 \Gamma}, \cdots$$

1 or $\rho = 1 + \vec{p}\cdot\vec{\sigma}$

$$D^{j_1\star}_{\lambda_1,\lambda_1'}(\alpha,\beta,\gamma)$$

alignment

probability: $|\mathcal{A}|$^2

Decay Group: $\mathcal{A} = \tilde{A}_1 + \tilde{A}_2 + \cdots$

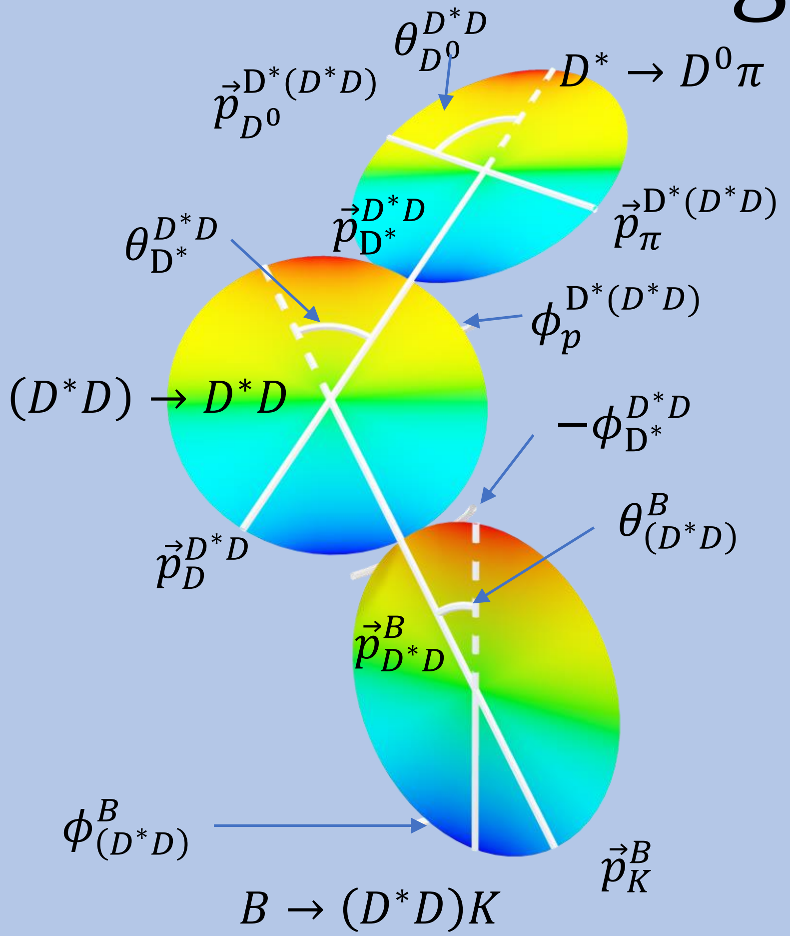
Decay Chain: $\tilde{A} = A_1 R A_2 \cdots$

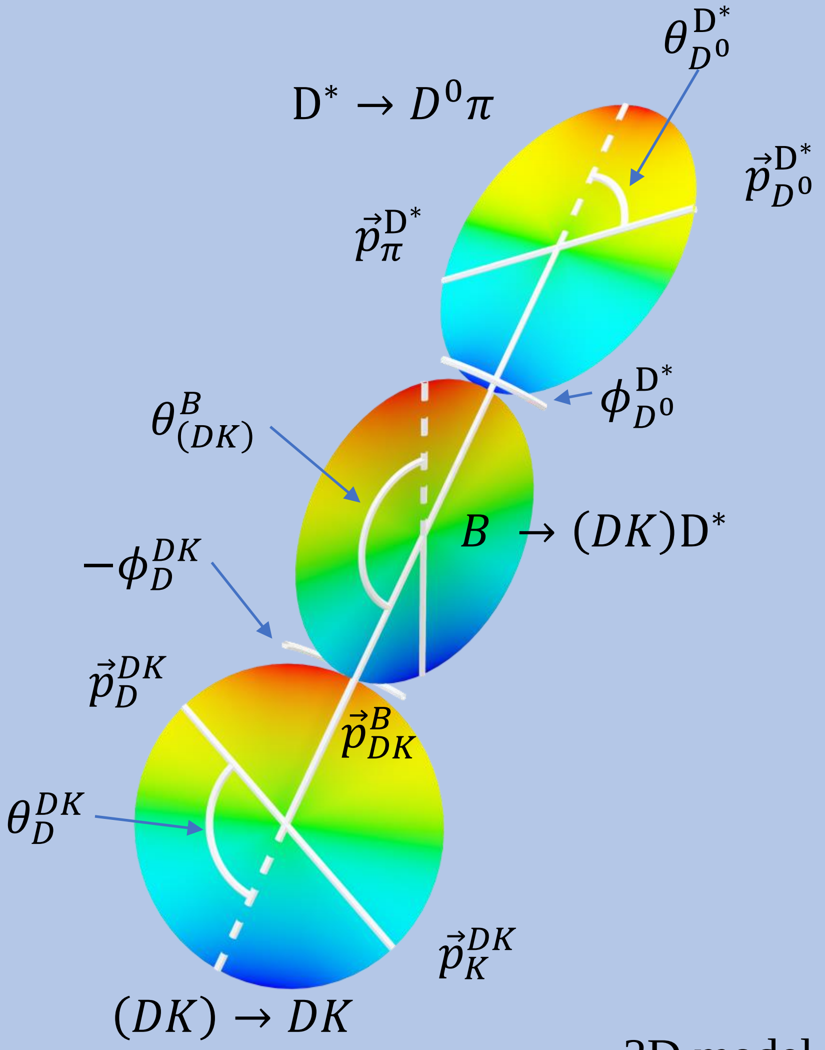
Decay: Wigner D-matrix, $A = HD^{*J}(\phi,\theta,0)$

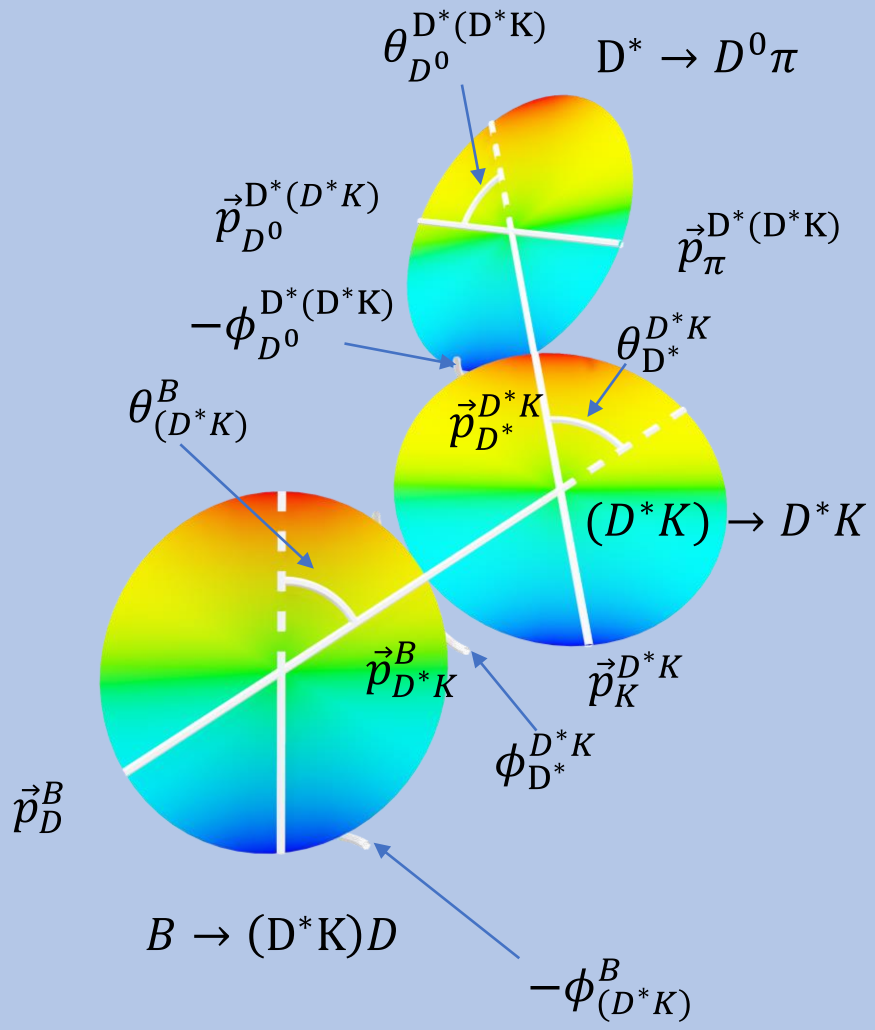
Particle: Breit-Wigner: $R(m)$, user defined

Automatic Angle Plot

$\vec{p}_B^{A}$ means momentum of B in the rest frame of A
ϕ means the rotation is anticlockwise, while $-\phi$ for clockwise
The sign is dependent on data







Auto calculated by TF-PWA,
Only required the 4-monmenta

TF-PWA also provide reverse process:
Mass + helicity angle -> 4-monmenta

3D model generated by a script using TF-PWA.

Custom Model

Line: $R(M;a) = M + a$

```python
from tf_pwa.amp import register_particle
from tf_pwa.amp import Particle

@register_particle("Line")
class LineModel(Particle):

  def init_params(self):  # define parameters
    self.a = self.add_var("a")

  def get_amp(self, *args, **kwargs):
    """ model as m + a ""“
    # write code with TF
    m = args[0]["m"]
    zeros = tf.zeros_like(m)
    return tf.complex(m + self.a(), zeros)
```

Define a custom model is simple.

$$H_{[\lambda_R\lambda_B]}(x;\vartheta)D^{j_A\star}_{[\lambda_A,\lambda_R-\lambda_B]}(x)R\,(x;\vartheta)H_{[\lambda_C,\lambda_D]}(x;\vartheta)D^{j_R\star}_{[\lambda_R,\lambda_C-\lambda_D]}(x)$$
$$D^{j_B\star}_{[\lambda_B,\lambda_B']}(x)D^{j_C\star}_{[\lambda_C,\lambda_C']}(x)D^{j_D\star}_{[\lambda_D,\lambda_D']}(x) \to A_{[\lambda_A,\lambda_B',\lambda_C',\lambda_D']}(x;\vartheta)$$

$$\lambda_{[RB][ARB][\,][CD][RCD][BB'][CC'][DD'] \to [AB'C'D']}$$

The shape is (number of events,), type is complex128

$R(x;\vartheta)$
- x: all data, (*args, **kwargs)
- ϑ: all parameters (self.a , ⋯)

here the data, (*args, **kwargs) is passed from DecayChain.

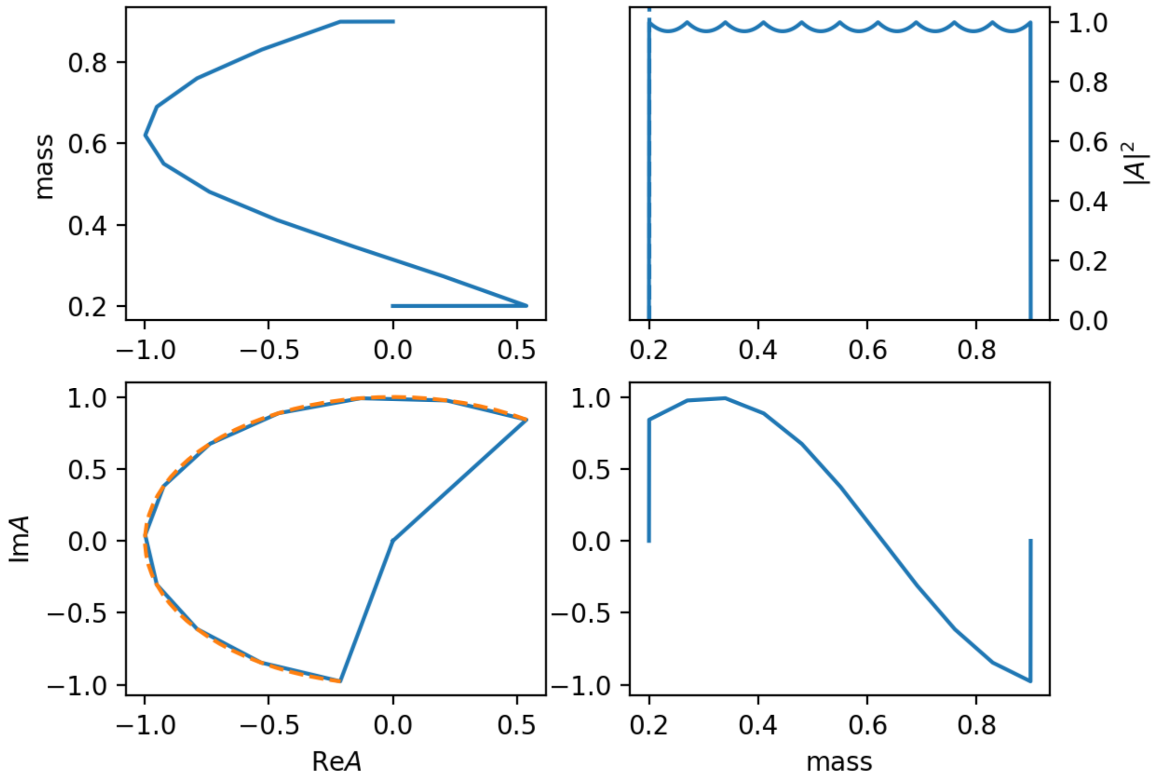
For convenience, different data will be divided into different parts to pass to get_amp(self, *args, **kwargs)

The parameters are directly defined in the class, and the values are obtained from VarsManager.

Use register_particle to register it,
then it can be used in config.yml

Implement of theoretical model

11 points leaner interpolation of $\exp(5im)$
The out of range is set to zeros

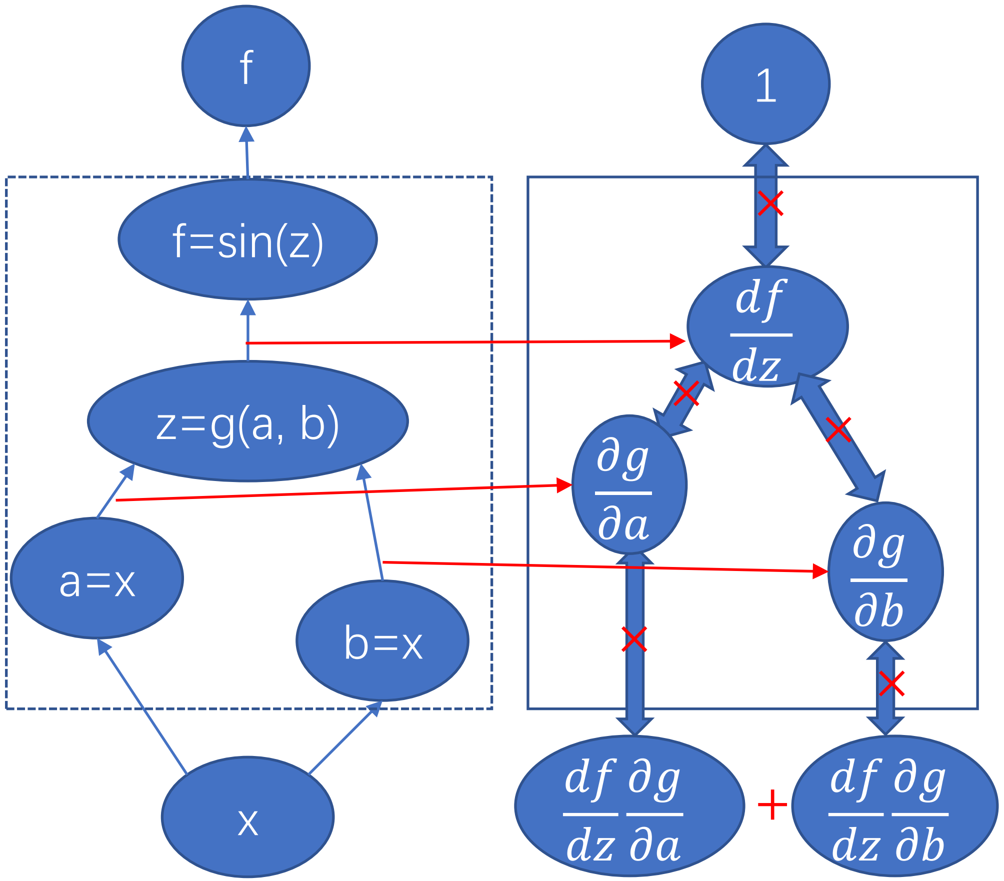


- Interpolation
  - Use for mass dependent model
  - Linear interpolation model
    - "linear_txt"
    - Input point by point values
      - Point position
      - Real parts
      - Image parts
    - Fast evaluation without writing TF code
  - For example, Dispersion integral
- All value in input data
  - If model if much more complex, not only mass dependent
  - One can calculate the value for all data first and input with data
  - For example, Triangle Singularity

Automatic Differentiation(AD)

- Widely used in Optimization problem
  - Calculate gradient automatically
  - No need for exact formula.
  - Recorded the computation graphs.
    - Operator: function used (sin, g)
    - Intermediate value: (z= 1^1)
  - Mostly matrix form (Jacobian).
  - Just combine the operator (Chain Rules)
- Chain Rules
  - ×: $\frac{\partial f(g(x))}{\partial x_i} = \frac{df}{dg} \times \frac{\partial g}{\partial x_i}$
  - +: $\frac{\partial f(h(x), g(x))}{\partial x_i} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial x_i} + \frac{\partial f}{\partial g}\frac{\partial g}{\partial x_i}$

The same rule-based method as our amplitude calculation.



$$f = \sin(x^x) = \sin(z), z = x^x$$
$$g(a, b) = a^b$$

Automatic Differentiation (numerically):

$$\frac{df}{dx}(1) = \frac{df}{dz}(1^1)[\frac{\partial g}{\partial a}(1,1) + \frac{\partial g}{\partial b}(1,1)] = 0.5403$$

backward

AD in amplitude fit

- Minimize of $-\ln L\,(\vartheta)$
  - $-\frac{\partial \ln L}{\partial \vartheta}$ is the steepest descent direction, used by most of optimizer
  - Error matrix $V_{ij} = \left[-\frac{\partial^2 \ln L}{\partial \vartheta_i \partial \vartheta_j}\right]^{-1}$ can also be estimated though AD
  - AD advantage:
    - Automatic.
    - Fast estimation: Time cost for eval $-\ln L\,(\vartheta)$ and $-\frac{\partial \ln L}{\partial \vec{\vartheta}}$ are on the same level.
    - Accurate gradient: more stable results.
  - AD disadvantage:
    - Require well defined gradients (continuous).
      - Avoid step function, delta function.
    - Only support function with predefined gradients.
      - Use TensorFlow only, but also have an interface to define functions.
    - Large (GPU) memory cost for recording intermediate values.
      - Split data into small batches (discuss later).

AD for error propagation

- Error propagation
  - $\sigma_y^2 = \frac{\partial y}{\partial x}\sigma_x^2\frac{\partial y}{\partial x}$, $\frac{\partial y}{\partial x}$ can be calculated by AD
  - Simple interface (see right)
  - Example: uncertainties of fit fractions in TF-PWA
- Advance usage
  - Define function with gradient
    - AD + some part of numerical function
    - $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$, numerical: $\frac{\partial g}{\partial x} = \frac{g(x+\Delta x)-g(x-\Delta x)}{2\Delta x}$
    - Example: Obtain pole mass in TF-PWA (a iteration process)
  - Systematic uncertainties of fixed parameters (fixed mass and width)
    - $-\ln L = -\ln L(\vartheta, z)$, ϑ: fit parameters, z: fixed parameters
    - Minimum condition as implicit function
    - $-\frac{\partial \ln L}{\partial \vartheta} = 0 \Rightarrow \frac{\partial \vartheta_i}{\partial z} = -\left[\frac{\partial^2 \ln L}{\partial \vartheta_i \partial \vartheta_j}\right]^{-1}\frac{\partial^2 \ln L}{\partial \vartheta_j \partial z}$

```
with config.params_trans() as pt:
    # g1 is fixed to 1
    g2_r = pt["Lmdc->piz.Sigma(1385)p_g_ls_1r"]
    g2_phi = pt["Lmdc->piz.Sigma(1385)p_g_ls_1i"]
    alpha = 2*g2_r*tf.cos(g2_phi) / (1+g2_r*g2_r)
print(alpha, pt.get_error(alpha))
```

$$\alpha_{\Sigma(1385)\pi} = \frac{|H^{\Sigma(1385)}_{0,\frac{1}{2}}|^2 - |H^{\Sigma(1385)}_{0,\frac{-1}{2}}|^2}{|H^{\Sigma(1385)}_{0,\frac{1}{2}}|^2 + |H^{\Sigma(1385)}_{0,\frac{-1}{2}}|^2} = \frac{2\Re\left(g^{\Sigma(1385)}_{1,\frac{3}{2}} \cdot \bar{g}^{\Sigma(1385)}_{2,\frac{3}{2}}\right)}{|g^{\Sigma(1385)}_{1,\frac{3}{2}}|^2 + |g^{\Sigma(1385)}_{2,\frac{3}{2}}|^2}$$

```
decay = config.get_decay()
p = decay.get_particle("Sigma(1385)p")
with config.params_trans() as pt:
    pole = p.solve_pole()
    re = tf.math.real(pole)
    im = tf.math.imag(pole)
print((re, im), pt.get_error((re,im)))
```

AD in Large size of data

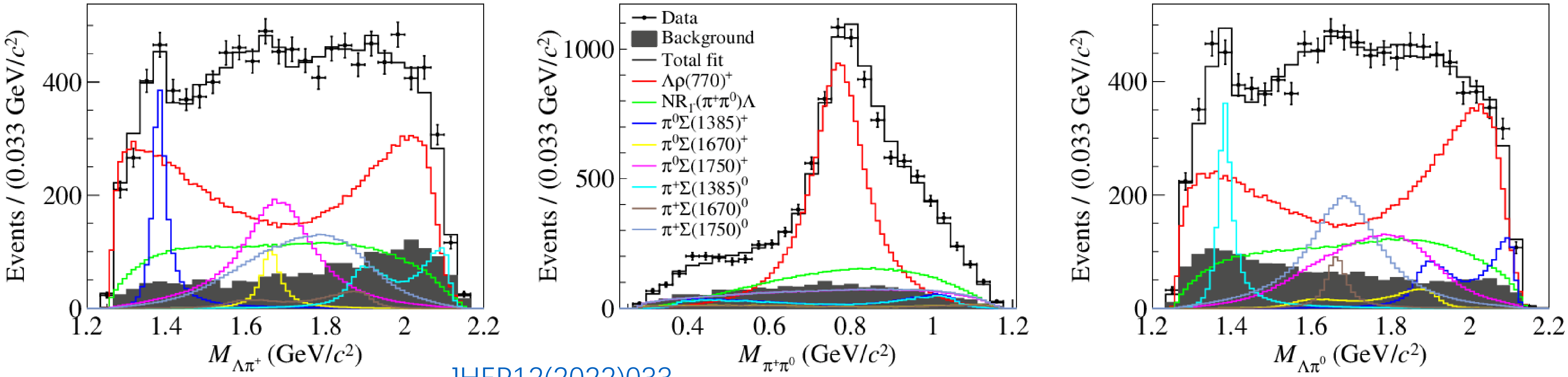
- Basic Log-Likelihood function
  - $\ln L(\vartheta) = \sum \ln \left[(1 - f_2) \frac{P_1(x;\vartheta)}{\int P_1(x;\vartheta)dx} + f_2 \frac{P_2(x;\vartheta)}{\int P_2(x;\vartheta)dx} \right]$
  - $I_i = \int P_i(x)dx \approx \sum \omega_j P_i(x_j)$
  - Required recording all $P_i(x_j)$ and intermediate values before gradients evaluations.
- Split large data into small batches (only record value in a small batch)
  - $\ln L(\vartheta) \Rightarrow \ln L(\vartheta; I_i)$
  - $\frac{\partial \ln L(\vartheta)}{\partial \vartheta} \Rightarrow \frac{\partial \ln L(\vartheta;I_i)}{\partial \vartheta} + \sum_i \frac{\partial \ln L(\vartheta;I_i)}{\partial I_i} \frac{\partial I_i}{\partial \vartheta}$
  - $\frac{\partial I_i}{\partial \vartheta} = \underbrace{\frac{\partial [\sum \omega_j P_i(x_j)]}{\partial \vartheta}}_{\text{Batch 1}} + \underbrace{\frac{\partial [\sum \omega_j P_i(x_j)]}{\partial \vartheta}}_{\text{Batch 2}} + \cdots$
- Expand the power of AD to multi GPU, even multi cluster

Example: fit results of $\Lambda_c^+ \to \Lambda\pi^+\pi^0$

- $\Lambda_c^+ \to \Lambda(\to p\pi^-)\pi^+\pi^0$
  - Simultaneous fit
    - 7 energy points
    - Total around 10k events, 854k MC
    - 38 free parameters
  - Dominated by $\Lambda_c^+ \to \Lambda\rho$: $57.2 \pm 4.2\%$
  - Clear peak for $\Lambda_c^+ \to \pi\Sigma(1385)$

Plot thought TF-PWA with simple config.yml
All decay chains will be added automaticly

```yaml
plot:
  mass:
    Sigma_star0:  # name in page 6
      display: "$M_{\\Lambda\\pi^{0}}$"
      bins: 30
      range: [1.2, 2.2]
      legend: False
```



JHEP12(2022)033

```yaml
data:
  …
  preprocessor: cached_shape
  amp_model: cached_shape
```

Different preprocess

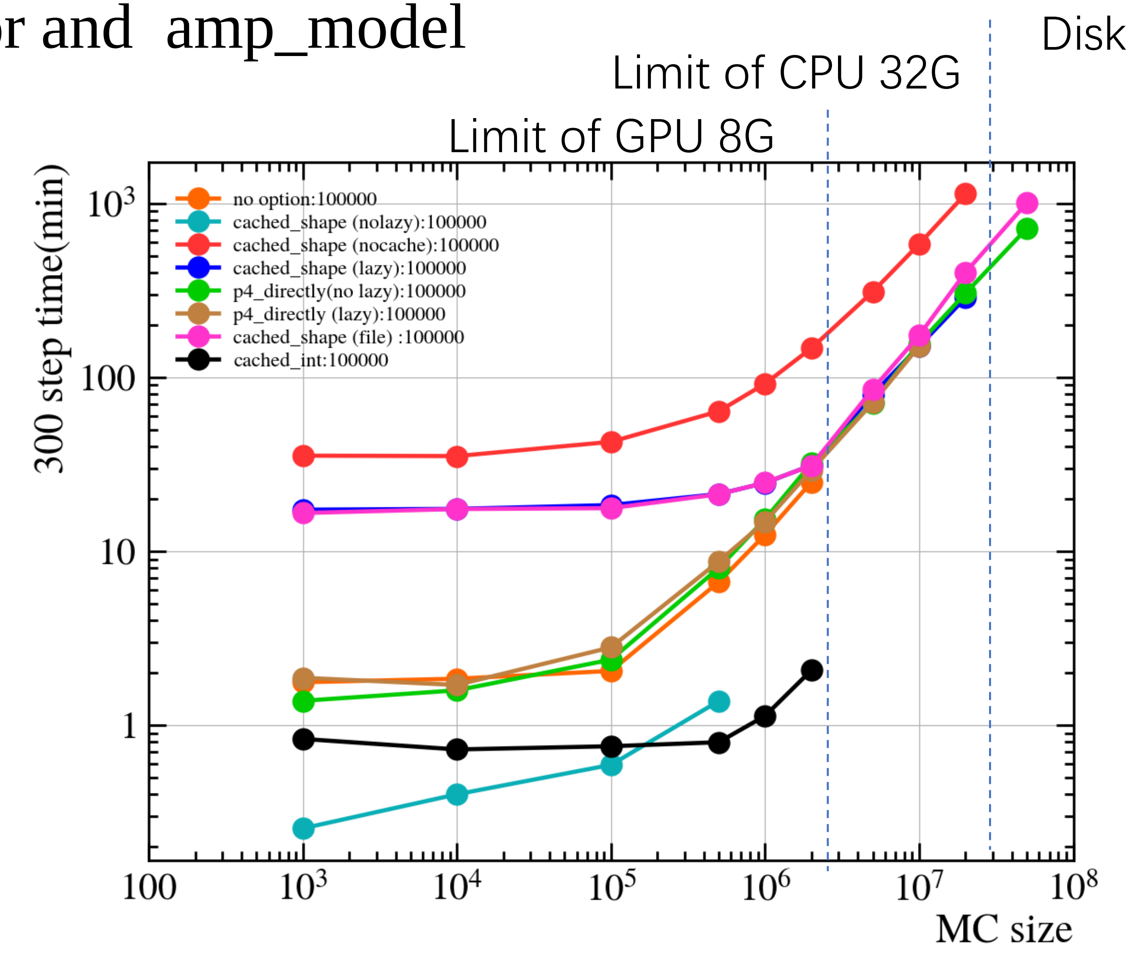
- Implement with different ways for amplitude
- Option in config.yml: data: preprocessor and amp_model

options	pre-process	amplitude
default	$p^\mu \to m, angle$	$c_i f_i(m) T_i(angle)$
cached_amp	$p^\mu \to m, angle, T_i$	$c_i f_i(m) T_i$
cached_shape	$p^\mu \to m, angle, T_i$ $f_j T_j$	$c_i f_i(m) T_i$ $c_j f_j T_j$
p4_directly	p^μ	$p^\mu \to m, angle$ $c_i f_i(m) T_i(angle)$

Table 1: Calculation in the two parts

data	memory requirement for one event
p^μ	4 N(particles)
m	N(particles)
$angle$	6 N(chain) N(decay in one chain) 3 N(chain) N(final particles)
$f_i T_i, T_i$	N(partial waves)N(helicity combination)

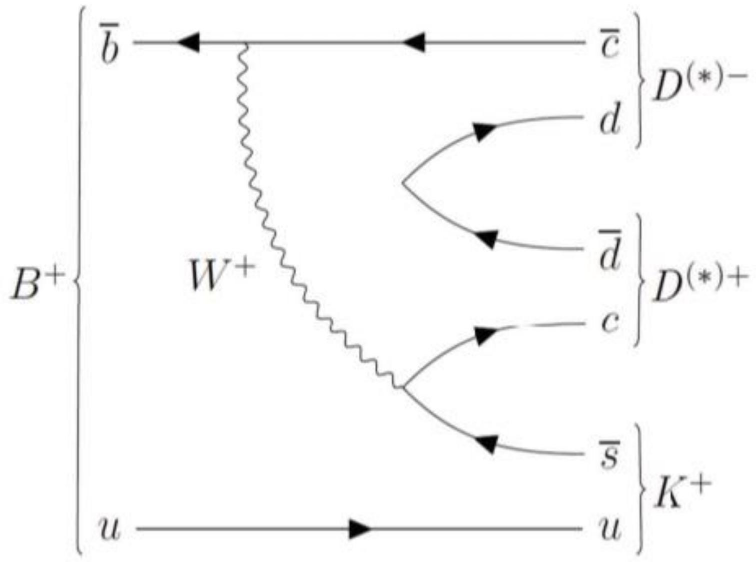
Table 2: Memory requirement for one event



$$B^{+} \to D^{*\pm} D^{\mp} K^{+}$$

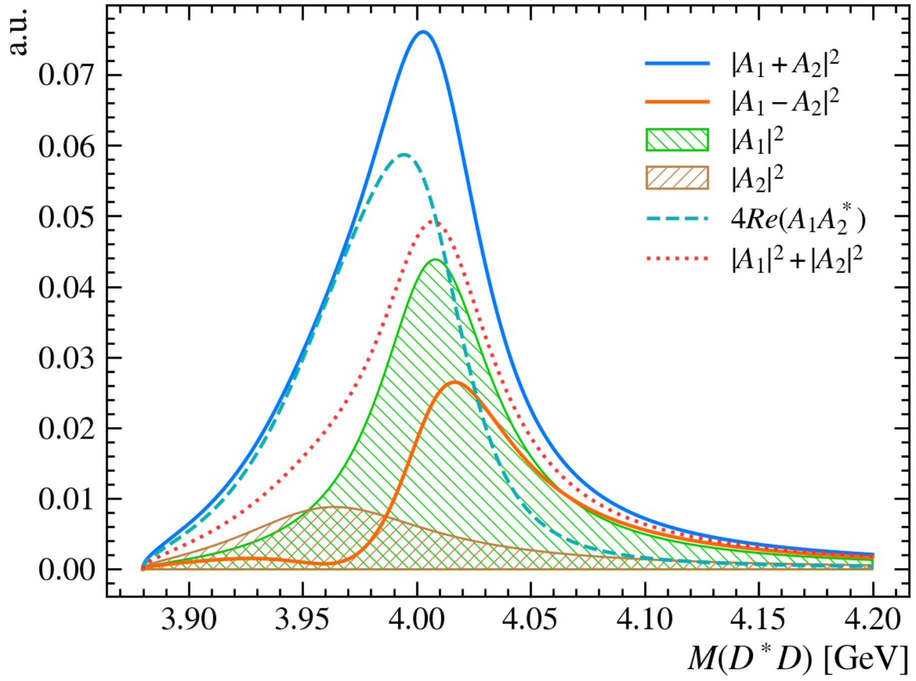
$B^+ \to D^{*\pm}D^{\mp}K^+$

- $B^+ \to D^{*\pm}D^{\mp}K^+$: two channels
  - $B^+ \to D^{*-}D^{+}K^+$
  - $B^+ \to D^{*+}D^{-}K^+$
- Possible resonances:
  - $D^{*\pm}D^{\mp}[c\bar{c}d\bar{d}]$: charmonium(-like)
    - 3.9 ~ 4. 8 GeV
    - Normal charmonium
      - $\psi(4040), \psi(4160), \chi_{c2}(3930), \chi_{c1}(4274)\cdots$
    - Charmonium(-like) states
      - $\chi_{c1}(3872)$ (aka $X(3872)$), $Z_c(3900), X(4020), Z_c(4430), \cdots$
  - $D^{(*)-}K^+[\bar{c}\bar{s}ud]$: $T^*_{cs0}(2870)^0$ (aka $X_0(2900)$), $T^*_{cs1}(2900)$(aka $X_1(2900)$)
    - Found in $B^+ \to D^+T^*_{cs0,1}$ ($\to D^-K^+$) **PRL 125 (2020) 242001, PRD 102 (2020) 112003**
  - $D^{(*)+}K^+[c\bar{s}u\bar{d}]$: $T^*_{c\bar{s}0}(2900)^{++}$
    - Found in $B^+ \to D^-T^*_{c\bar{s}0}(2900)^{++}(\to D_s^+\pi^+)$ **PRL 131 (2023) 041902, PRD 108 (2023) 012017**



C-parity relation

- Used to constrains $R^0 \to D^{*-}D^+$ and $R^0 \to D^{*+}D^-$
  - $A^{R^0 \to D^{*-}D^+} = C_R A^{R^0 \to D^{*+}D^-}$
  - C_R is the C parity of R
- Effect of this relation
  - Two contribution with the same J^P
    - $\mathrm{R}_1(C=+1)$ and $\mathrm{R}_2(\mathrm{C}=-1)$
  - For $B^+ \to R(\to D^{*+}D^-)K^+$
    - $|A|^2 = |A_1 + A_2|^2 = |A_1|^2 + |A_2|^2 + 2Re\,(A_1 A_2^*)$
  - For $B^+ \to R(\to D^{*-}D^+)K^+$
    - $|A'|^2 = |A_1 - A_2|^2 = |A_1|^2 + |A_2|^2 - 2Re\,(A_1 A_2^*)$
  - $|A|^2 - |A'|^2 = 4Re(A_1 A_2^*)$
    - Only the interference can contribute to the difference of two channel



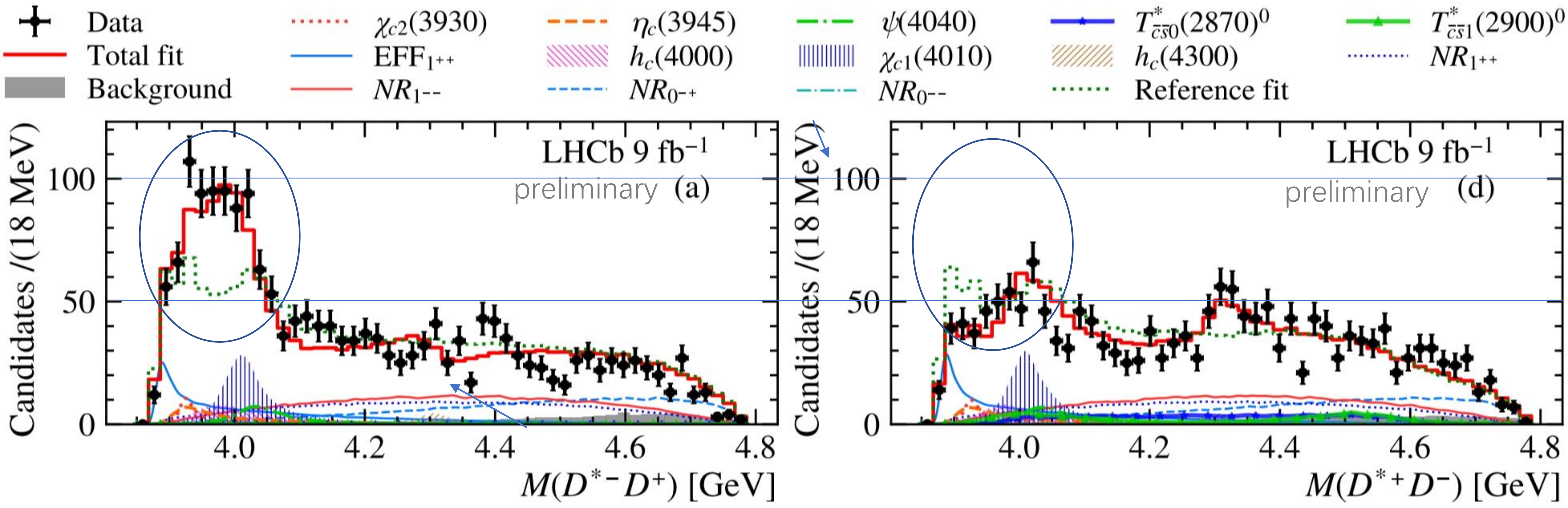
Implement with custom model in TFPWA

- Create new model based on original model
  - Simple inherited with original model
  - Override related parts

```
from tf_pwa.amp.core import register_particle, get_particle_model

def create_model(name): # common function to create new model
    cls = get_particle_model(name) # find the particle model with name

    @register_particle("C({})".format(name)) # register with “C(name)”
    class _NewClass(cls): # inherited form original model
        def get_amp(self, data, data_d, all_data=None, **kwargs): # override amplitude calculation
            d = all_data.get("c",1) # D*+D- or D*-D+
            if self.C == 1: # C parity is 1, use the same model
                 return super().get_amp(data, data_d, all_data=None, **kwargs)
            else: # C parity is -1, use amp for D*+D- and –amp for D*-D+
                amp = super().get_amp(data, data_d, all_data=None, **kwargs)
                return tf.where(d >0, amp, -amp)
    # other cases: DK, D*K …

create_model("BWR") # create new model “C(BWR)” for “BWR”
```

Charmuinon(-like) states

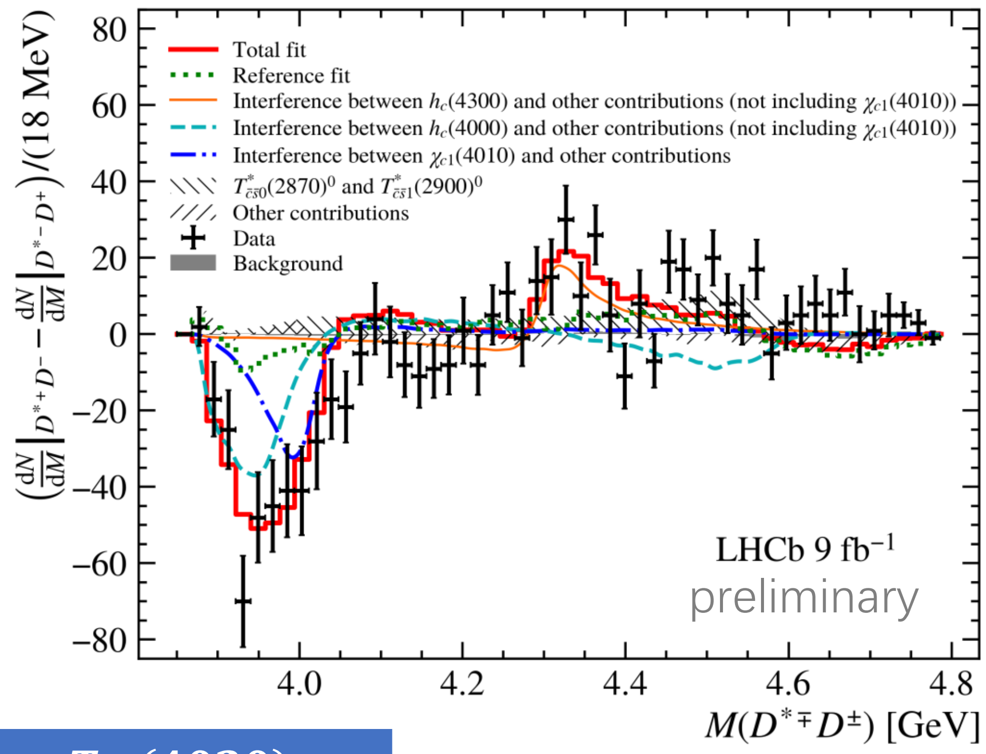
- Large difference in 3.9 – 4.0 GeV
  - $T^*_{cs0,1}$ is small in that range, do not expect large effect
  - Large difference due to C-parity



LHCb−PAPER−2023−047

Distribution of difference

- Difference
  - Require two new states with different C-parities
  - Prefer 1^{+} contribution in angular distribution.
- Reference fit w/o new states (green dotted)
  - Bad Fit quality
  - Required new states
- Add two states around 4.0 GeV
  - Similar mass as $T_{c\bar{c}}(4020)[1^{+-}]$ (aka $X(4020)$, $Z_c(4025)$)
  - $h_c(4000)[1^{+-}]$: much larger width than $T_{c\bar{c}}(4020)$
  - $\chi_{c1}(4010)[1^{++}]$: different C-parity than $T_{c\bar{c}}(4020)$



	$h_c(4000)$	$\chi_{c1}(4010)$	$T_{c\bar{c}}(4020)$
J^{PC}	1^{+-}	1^{++}	1^{+-}
$mass$/MeV	4000^{+17+29}_{-14-22}	$4012.5^{+3.6+4.1}_{-3.9-3.7}$	$4025^{+2.0}_{-4.7} \pm 3.1$
$width$/MeV	184^{+71+97}_{-45-61}	$62.7^{+7.0+6.4}_{-6.4-6.6}$	$23.0 \pm 6.0 \pm 1.0$

PRL 115 (2015) 182002

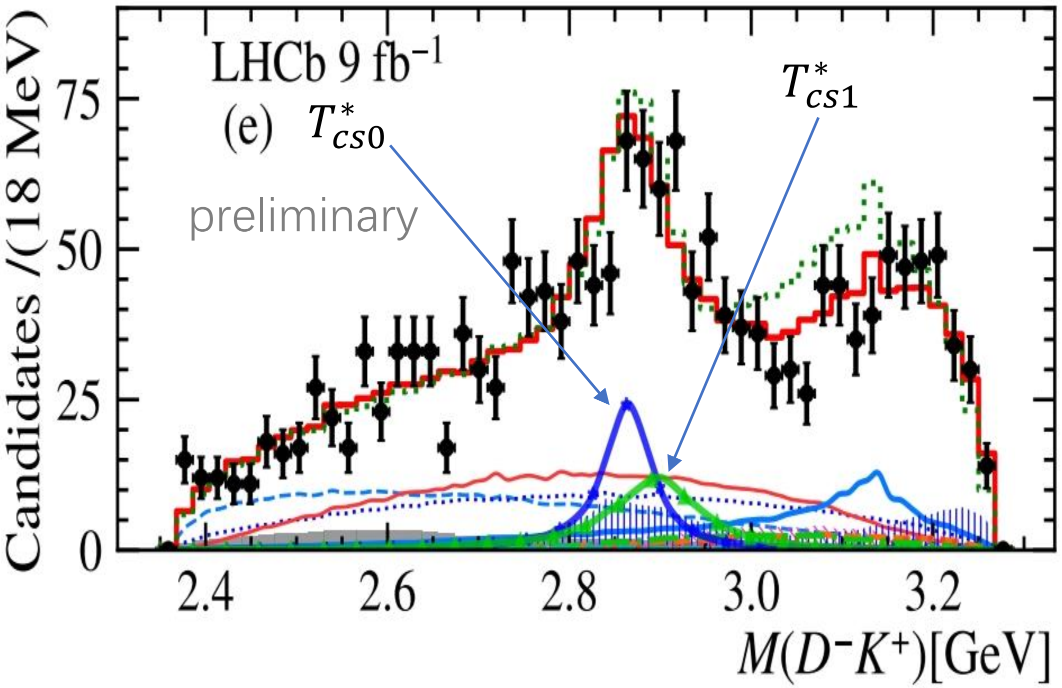
LHCb−PAPER−2023−047

$T^*_{cs0}(2870)^0$ and $T^*_{cs1}(2900)^0$

PRL 125 (2020) 242001, PRD 102 (2020) 112003

- Previous work
  - $B^+ \to XD^+, X \to D^-K^+$
    - $B^+ \to T^*_{cs0}(2870)^0 D^+$: S wave
    - $B^+ \to T^*_{cs1}(2900)^0 D^+$: only P wave
- New production mode
  - $B^+ \to XD^{*+}, X \to D^-K^+$
    - $B^+ \to T^*_{cs0}(2870)^0 D^{*+}$: P wave
    - $B^+ \to T^*_{cs1}(2900)^0 D^{*+}$: S, P, D waves
- Different ratios of branching factions

	$B^+ \to D^{*+}D^-K^+$	$B^+ \to D^+D^-K^+$
$\frac{B(B^+ \to T^*_{cs0}(2870)^0 D^{(*)+})}{B(B^+ \to T^*_{cs1}(2900)^0 D^{(*)+})}$	$1.17 \pm 0.31 \pm 0.48$	0.18 ± 0.05



- Mass and width consistent within uncertainties

Summary

- TF-PWA
  - Convenient configuration, general proposed
  - Easy to implement new models
  - Use powerful AD in fitting and error propagation.
  - Provide options to achieve high performance
- $B^+ \to D^{*\pm} D^{\mp} K^+$
  - C-parity relation
  - Observe new charmonium(-like) states
  - Confirm $\mathrm{T}_{cs0}^{*}(2870)^0$, $\mathrm{T}_{cs1}^{*}(2900)^0$ in new production mode

Thank you for your attentions!

Backup

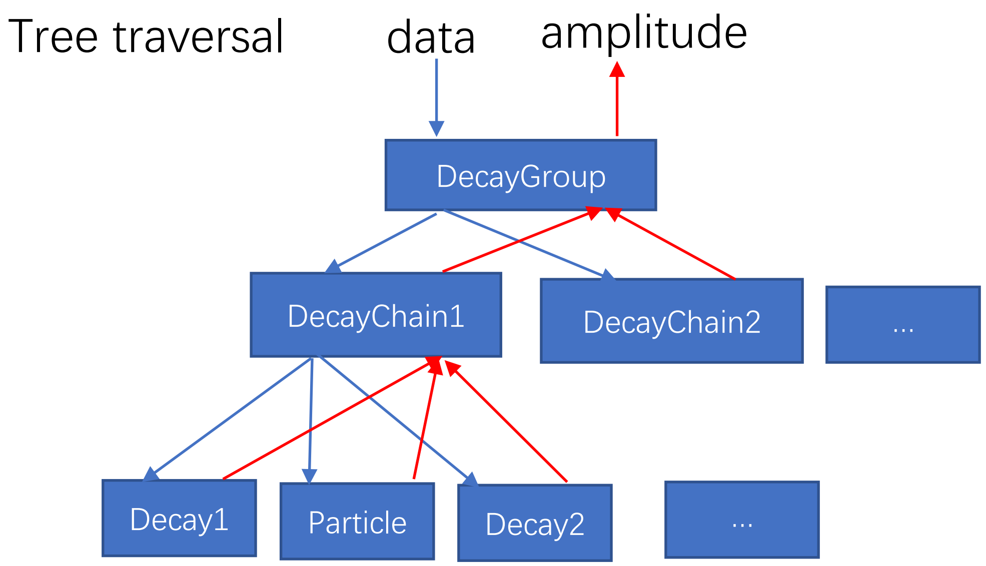
-

Topology of decay chain

- Define
  - All combination of final particles
- Same
  - Topo: (D, E), (C, D, E)
  - A->R1 +B ,R1 -> R2 + C, R2 -> D + E
  - A->R3 +B ,R4 -> R5 + C, R5 -> D + E
- Not same
  - Topo: (B, C), (D, E)
  - A->R6 +R7 ,R6 -> B + C, R7 -> D + E

Implements

- Class structure
  - Two main method
    - init_params()
      - Define fit parameters
    - get_amp(data) -> amp
      - pass data to substructures
      - Use fit parameters and data to calculate amplitude
- Einstein summation convention
  - $A_{abdf} = \sum_c A_{abc} A_{cdf}$: $abc, cdf \to abdf$
  - The index for each decay is well-known: the helicity of final particles.
  - Use in Decay Chain, combine all parts together.
- Global model list
  - register_particle(name)
    - Write model into a global list
    - New model can inherit from origin model
  - build with YAML
    - use the name to find it, and create with args.



probability: $|\mathcal{A}|$^2

Decay Group: $\mathcal{A} = \tilde{A}_1 + \tilde{A}_2 + \cdots$

Decay Chain: $\tilde{A} = A_1 R A_2 \cdots$

Decay: Wigner D-matrix, $A = HD^{*J}(\phi, \theta, 0)$

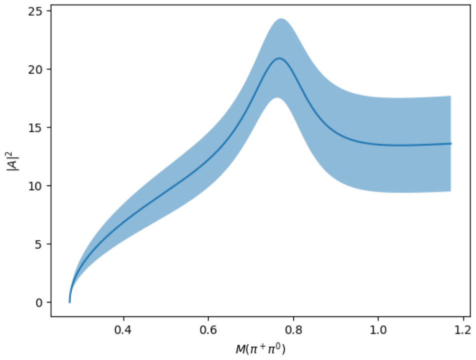
Particle: Breit-Wigner: $R(m)$, user defined

Amplitude as a function

- Reverse process of angle calculation
  - Mass + Helicity angle -> 4- momenta
  - $(m_0, \phi_0, \theta_0, m_{12}, \phi_{12}, \theta_{12}) \xrightarrow{\text{transform}} p_1^\mu, p_2^\mu, p_3^\mu$
- Factor system:
  - Eval amplitude of special partial wave though control of parameters
  - $A(p_1^\mu, p_2^\mu, p_3^\mu) \xrightarrow{g_{i\neq j}=0} g_i A_i(p_1^\mu, p_2^\mu, p_3^\mu)$
- Combine Together: Lineshape function of special wave
  - Set angle to 0, $D_{m,m'}(0,0,0) = \delta_{m,m'}$ is constant.
  - Vary mass, then get the shape of masses
  - $f(m_{12}) = g_i A_i\ (m_0, \phi_0 = 0, \theta_0 = 0, m_{12}, m_{12}, \phi_{12} = 0, \theta_{12} = 0)$
  - No worries for the complex formula
- 2D function of amplitude
  - 2D plot Dalitz variables
  - 2D plot of Mass and cos θ

```
f1 = config.get_particle_function("R1")
f2 = config.get_particle_function("R2")

m = f1.mass_linspace(1000)
# plot the first wave
amp = tf.abs(f1(m)[:,0] + f2(m)[:,0])**2
plt.plot(m, amp)
```

Uncertainties from error propagation



Process of data

```
data:
  dat_order: [B, C, D]
  data: [data.dat]
  phsp: [phsp.dat]
```

- Input
  - p_i^μ of final particles directly
    - dat_order is the order of p_i^μ
    - data, phsp (and bg) are used for input data files
      - plain text file for p_i^μ (E px py pz)
  - Additional information
    - Suffix for data, phsp (and bg)
      - Weight for weights
      - Charge for charges
      - …
    - Support number for all events or a file for each events
- Preprocess:
  - Automatic angle calculation
    - Tree traversal for data structure
    - Euler angle from coordinates defined by momentums
    - With alignment angle (see backup)
    - Use topology structure to reduce costs of memory

Algorithm 1: Automatic angle calculation

Input: initial coordinate axis $\vec{z_0}$ and $\vec{y_0}$, momentum after a chain boost

Output: angle θ, ϕ of all particles

```
set initial particle data: {(z⃗0, y⃗0), L0 = 1}
for decay from top to finals (pre-order traversal) do
    for particles a that product though decay do
        boost p_a^μ to the rest frame of the decay.
    end
    for k-th out particle (b) in the decay do
        {(z⃗0, y⃗0) L0} is data for input particle.
        p⃗ is the direction of b in the rest frame.
        z⃗ = p⃗/|p⃗|, y⃗ = z⃗0 × z⃗/|z⃗0 × z⃗|.
        output: θ = arccos(z⃗0 · z⃗).
        set the range of φ to [−kπ, 2π − kπ],
        output: φ = atan2(z⃗0 · (y⃗0 × y⃗), y⃗0 · y⃗).
        if mass of b is not 0 then
            ω = tanh^{-1}(1/√(1 − p²/E²))
        else
            ω = sinh^{-1}((E² − 1)/(2E))
        end
        L = B_z(ω) R_y(θ) R_z(φ) L0.
        set particle b data: {(z⃗, y⃗), L}
    end
end
for k-th final particle do
    L = L_{ref,k} L_k^{-1}
    α = arg L22 − arg L21
    β = cos^{-1}(L11 L22 + L21 L21)
    γ = arg L22 + arg L21
end
```

Formula of Resolution

- Detector: Combine effect of resolution and efficiency
- An event x was detected as y with probability
  - $p(x \to y) = \epsilon_x(x) R_x(x \to y)$
- The real probability of y: $p(y) = \int |A|^2(x) p(x \to y) dx$

$$\epsilon_y(y) = \int \epsilon_x(x) R_x(x \to y) dx, R_y(x \to y) = \frac{p(x \to y)}{\epsilon_y(y)}$$

$$p(y) = \int |A|^2(x) p(x \to y) dx = \epsilon_y(y) \int |A|^2(x) R_y(x \to y) dx$$

- $\epsilon_y(y)$ can be obtained by Phase Space MC. The distribution of reconstructed variables
- $R_y(x \to y)$ in normalize probability function that y from all possible x.
- Use $R_y(x \to y)$ to do the convolution,
  - Use phase space MC for flat x, then $R_y(x \to y)$ is the projection of $p(x, y)$ with fixed y
  - Normalized $\int R_y(x \to y) dx = 1$.
- Use MC truth to do the integration
  - $\int |A|^2(x) \int \epsilon_y(y) R_y(x \to y) dy dx = \int |A|^2(x) \epsilon_x(x) dx$.

Alignment angle in TFPWA for helicity formula

$Boost: B_z(\omega), Rotation: R_z(\phi), R_y(\theta)$

For final state $|out\rangle = |p_1\rangle \otimes |p_2\rangle \otimes |p_3\rangle$,choose a single particle state $|p_1\rangle$.
The final state define in $0 \to R, 2; R \to 1, 3$:

$$|p_1\rangle_R = B_z(\omega_1)R_z(0)R_y(\theta_1)R_z(\phi_1)|p_1\rangle = L_1|p_1\rangle$$
$$|p_1\rangle_1 = B_z(\omega_2)R_z(0)R_y(\theta_2)R_z(\phi_2)|p_1\rangle = L_2|p_1\rangle_R$$

On the other decay chain $0 \to R', 3; R' \to 1, 2$:

$$|p_1\rangle_{R'} = B_z(\omega_1')R_z(0)R_y(\theta_1')R_z(\phi_1')|p_1\rangle = L_1'|p_1\rangle$$
$$|p_1\rangle_2 = B_z(\omega_2')R_z(0)R_y(\theta_2')R_z(\phi_2')|p_1\rangle = L_2'|p_1\rangle_{R'}$$

The finals state is the rest frame, so no boost remained.
The alignment angle is the rotation

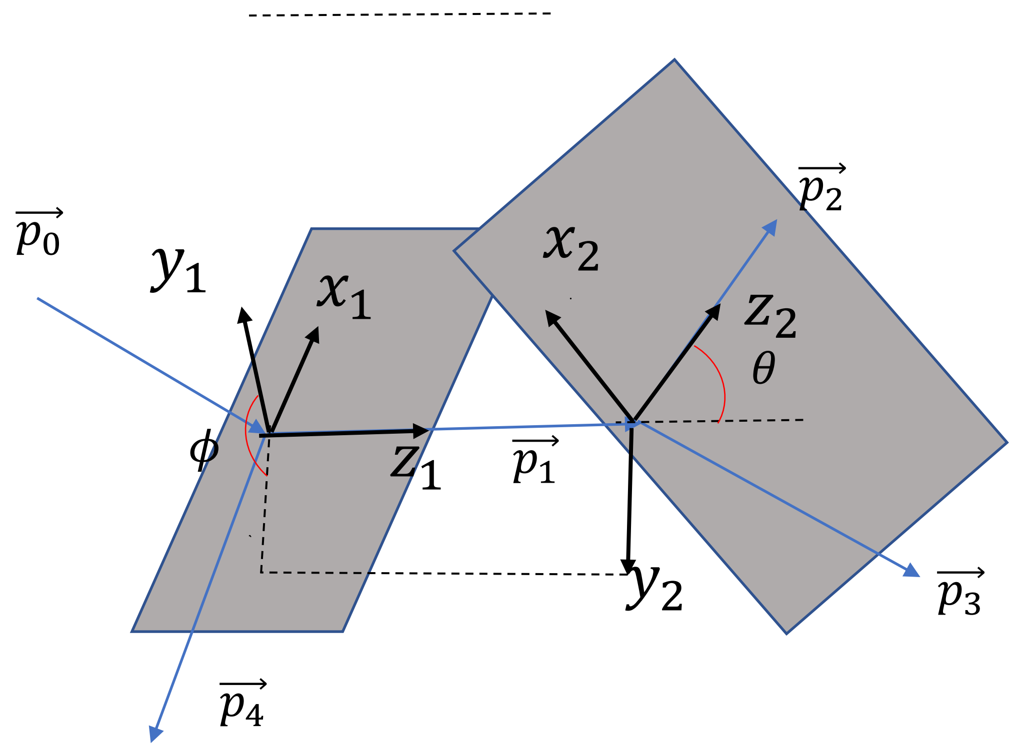
$$|p_1\rangle_2 = L_r|p_1\rangle_1 = R_z(\gamma)R_y(\beta)R_z(\alpha)|p_1\rangle_1$$

So

$$L_r = R_z(\gamma)R_y(\beta)R_z(\alpha) = L_2'L_1'L_1^{-1}L_2^{-1}$$

In general

$$L_r = L_a L_b^{-1} = \left(\prod_i L_{n-i}'\right)\left(\prod_j L_j^{-1}\right)$$



choose SU(2) representation as

$$\omega = \operatorname{arccosh}\frac{1}{\sqrt{1-\frac{v^2}{c^2}}} \qquad B_z(\omega) = \begin{pmatrix} e^{-\frac{\omega}{2}} & 0 \\ 0 & e^{\frac{\omega}{2}} \end{pmatrix}, R_z(\phi) = \begin{pmatrix} e^{-\frac{i\phi}{2}} & 0 \\ 0 & e^{\frac{i\phi}{2}} \end{pmatrix}, R_y(\theta) = \begin{pmatrix} \cos\frac{\beta}{2} & -\sin\frac{\beta}{2} \\ \sin\frac{\beta}{2} & \cos\frac{\beta}{2} \end{pmatrix}$$

$$L_{ab} = R_z(\gamma)R_y(\beta)R_z(\alpha) = \begin{pmatrix} \cos\frac{\beta}{2}e^{-\frac{i(\alpha+\gamma)}{2}} & -\sin\frac{\beta}{2}e^{\frac{i(\alpha-\gamma)}{2}} \\ \sin\frac{\beta}{2}e^{-\frac{i(\alpha-\gamma)}{2}} & \cos\frac{\beta}{2}e^{\frac{i(\alpha+\gamma)}{2}} \end{pmatrix}$$

$$\cos\beta = \cos^2\frac{\beta}{2} - \sin^2\frac{\beta}{2} = L_{11}L_{22} + L_{12}L_{21}, \beta \in [0,\pi]$$

$$\alpha+\gamma = -2\operatorname{ang}L_{11} = 2\operatorname{ang}L_{22}, \ \alpha-\gamma = -2\operatorname{ang}L_{12} = -2\operatorname{ang}L_{21}$$

$$|L_{ab}| = 1 \qquad L_{ab}^{-1} = \begin{pmatrix} L_{22} & -L_{12} \\ -L_{21} & L_{11} \end{pmatrix}$$

Simple AD implement

- backward AD

```
Var a = Var(3.1415926);
auto b = SinOp(&a);
auto c = AddOp(&a, &b);
c.backward(1.0);
std::cout<< a.grad;
```

$x + \sin x$

$grad(Add, Var)$

$grad(Var, x)$

$+grad(Add, \sin(x))$

$grad(\sin(x), Var)$

$grad(\mathrm{Var}, \mathrm{x})$

Output: 1.44329e-15 $\approx 1 + \cos \pi$

1. have to use defined Op
2. caching forward results,
   1. improve the speed
   2. more memory required
3. Vectorized:
   1. single operator, multiple data
   2. optimized for linear algebra

```
#include<cmath>
#include<vector>
class Op {
    public: std::vector<Op*> inputs;
    virtual double forward() = 0;
    virtual void backward(double grad=1) = 0;
};
class Var: public Op {
    public: double value, grad;
    Var(double value): value(value), grad(0.) {inputs={};};
    double forward() override { return value;}
    void backward(double grad=1) override {
         this->grad += grad;}
};
class SinOp: public Op {
public: SinOp(Op* input) {inputs = {input};};
    double forward() override {
         return sin(inputs[0]->forward());}
    void backward(double grad=1) override {
        inputs[0]->backward(grad * cos(inputs[0]->forward()));
    }
};
class AddOp: public Op {
     public: AddOp(Op* x, Op* y) {inputs = {x, y};};
     double forward() override {
         return inputs[0]->forward() + inputs[1]->forward(); }
     void backward(double grad=1) override {
        inputs[0]->backward(grad);
        inputs[1]->backward(grad);
    }
};
```

Likelihood formula

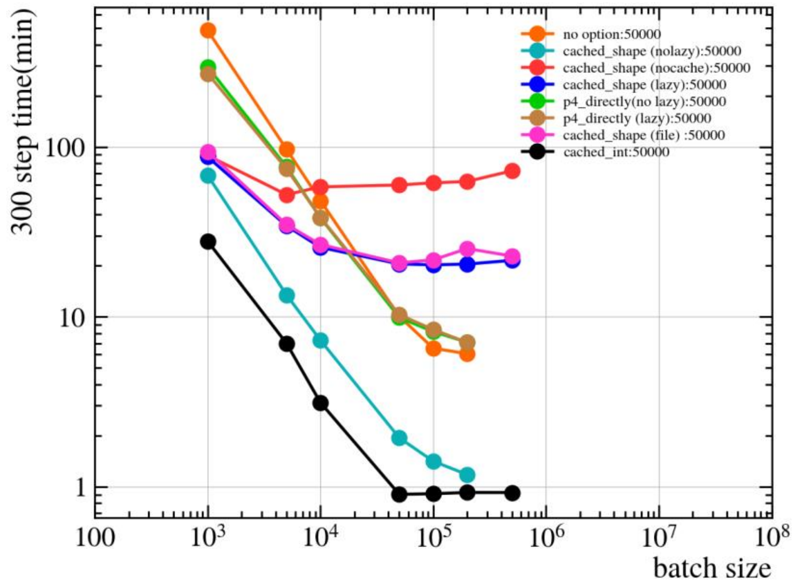
```yaml
data:
  dat_order: [B, C, D]
  data: [data.dat]
  bg: [bg.dat]
  bg_weight: 0.1
  phsp: [phsp.dat]
```

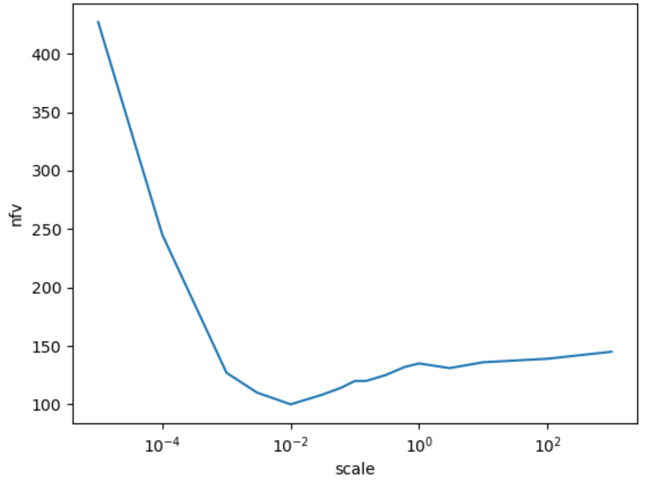
- Option in config.yml:data: model
- Default
  - $-\ln L = -\sum_{i\in data} w_i \ln|A|^2(x_i) + (\sum_{i\in data} w_i) \ln I_{sig}$
  - $I_{sig} = \frac{\sum_{j\in MC} \omega_j |A|^2(x_j)}{\sum_{j\in MC} \omega_j}$
  - bg will be merged into data with -weights
- cfit:
  - Additional information data:
    - bg_value for value of bg
  - Additional config
    - bg_frac: f_{bg}
  - -ln $L(\vartheta) = -\sum \ln\left[(1-f_{bg})\frac{|A|^2(x;\vartheta)}{I_{sig}} + f_{bg}\frac{B(x;\vartheta)}{I_B}\right]$

```yaml
data:
  dat_order: [B, C, D]
  model: cfit
  bg_frac: 0.1
  data: [data.dat]
  data_bg_value: [data_bgv.dat]
  phsp: [phsp.dat]
  phsp_bg_value: [phsp_bgv.dat]
```

Other option affecting fit time

- Bacth size : config.fit(batch=n)
  - Batch for calculating gradient
  - Large is better but required large memory
- tf.function: use_tf_function
  - Compile function to reduce python operation
    - Add no_id_cached: True when use lazy_call
  - Additional jit_compile
  - Required addition memory and setup time
- Grad scale: config.fit(grad_scale=x)
  - Can reduce iterations to best minimal





Batch for Hessian

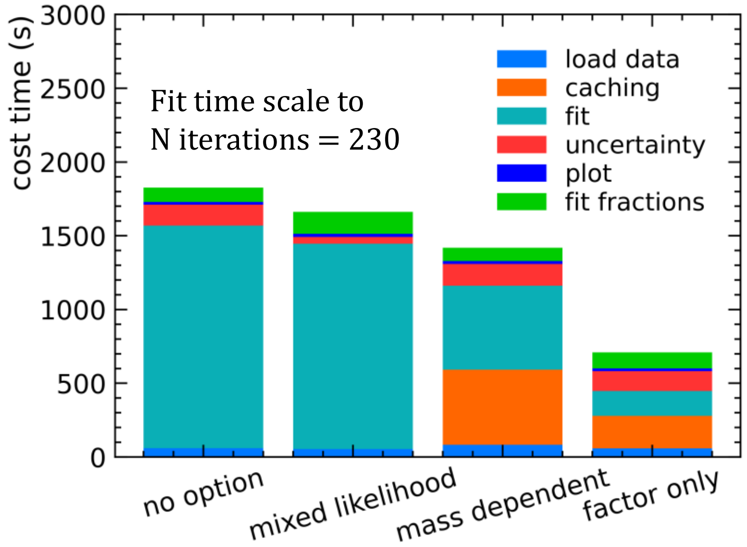
- $\frac{\partial \ln L(\vartheta)}{\partial \vartheta} = \frac{\partial \ln L(\vartheta, I)}{\partial \vartheta} + \frac{\partial \ln L(\vartheta, I)}{\partial I}\frac{\partial I}{\partial \vartheta}$
- $\frac{\partial^2 \ln L(\vartheta)}{\partial \vartheta \partial \vartheta} = [\frac{\partial^2 \ln L(\vartheta, I)}{\partial \vartheta \partial \vartheta} + \frac{\partial^2 \ln L(\vartheta, I)}{\partial \vartheta \partial I}\frac{\partial I}{\partial \vartheta}] + (\frac{\partial^2 \ln L(\vartheta, I)}{\partial I \partial \vartheta}\frac{\partial I}{\partial \vartheta} + \frac{\partial^2 \ln L(\vartheta, I)}{\partial I \partial I}\frac{\partial I}{\partial \vartheta}\frac{\partial I}{\partial \vartheta}) + \frac{\partial \ln L(\vartheta, I)}{\partial I}\frac{\partial^2 I}{\partial \vartheta \partial \vartheta}$
- Step1: eval $(I, \frac{\partial I}{\partial \vartheta}, \frac{\partial^2 I}{\partial \vartheta \partial \vartheta})$ in small batch
- Step2: eval $(\ln L(\vartheta'), \frac{\partial \ln L(\vartheta')}{\partial \vartheta'}, \frac{\partial \ln L(\vartheta')}{\partial \vartheta' \partial \vartheta'})$ in small batch
  - Here: $\vartheta_i' = (\vartheta_i, I), \frac{\partial \vartheta_i'}{\partial \vartheta_j} = (\delta_{ij}, \frac{\partial I}{\partial \vartheta_j})$
- Step3: $\frac{\partial^2 \ln L(\vartheta)}{\partial \vartheta_i \partial \vartheta_j} = \frac{\partial^2 \ln L(\vartheta')}{\partial \vartheta_k' \partial \vartheta_l'}\frac{\partial \vartheta_k'}{\partial \vartheta_i}\frac{\partial \vartheta_l'}{\partial \vartheta_j} + \frac{\partial^2 \ln L(\vartheta')}{\partial I}\frac{\partial^2 I}{\partial \vartheta_i \partial \vartheta_j}$

Real analysis performance

Environment:
NVIDIA RTX 3080
TensorFlow 2.2
CUDA 10.1

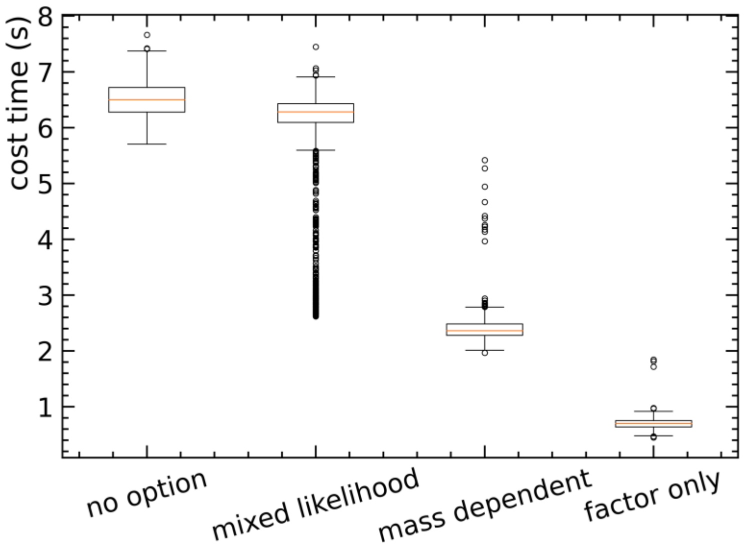
- Optimized method in Factor System page
  - Caching method
    - Large time for caching
    - required more memory
    - limited to special cases
  - All the process is automatic (from config.yml to all basic results)

Total fit time



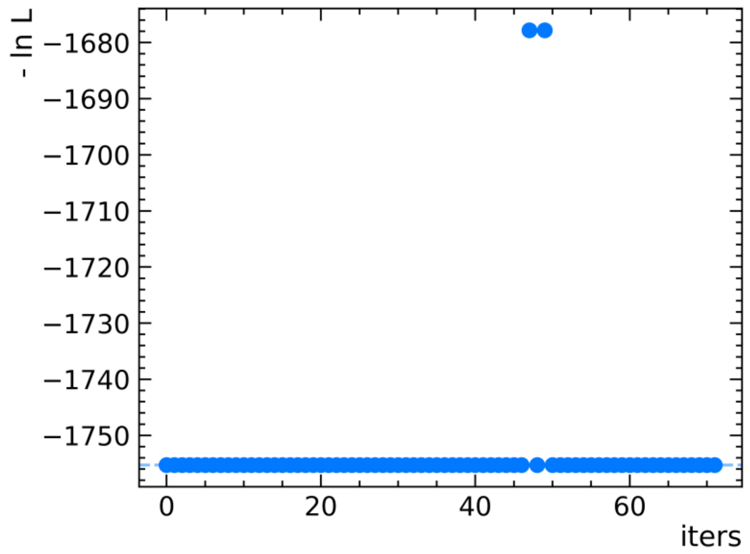
Only half of hours
for ALL the process

Time in each iterations



Caching provide 8 times speed up
for fit parts

Fit stability test



Fit stability looks well
in random initial parameters

Factor system: automatic factorization of amplitude

- Amplitude can be written as the combination of summation and production.

$$A = (\sum g_i A_i)(\sum g_j' A_j') \cdots \Rightarrow A = \sum_{ij} (g_i g_j')(A_i A_j')$$

$$G_{(i,j)} = g_i g_j' = \begin{cases} 1 & i,j = (a,b) \\ 0 & i,j \neq (a,b) \end{cases} \Rightarrow A = B_{(i,j)} = A_i A_j'$$

- No need known for the exact formula A_i, just use the parameters g_i.
- Some special treatment is implemented as option for better performances. (comparing in Page 21)
  - Amplitude caching method
    - mass dependent:

      $A(p_i^\mu) = \sum g_i R(m) A_i(p_i^\mu) \Rightarrow A(m) = \sum g_i R(m) B_i$

    - factor only: (only for MC integration)
      - $\sum |A|^2 \to G_i G_j^* (\sum C_{ij})$
      - Required all shape parameters fixed
  - Special in simultaneous fit
    - Mixed likelihood, avoid small size data

Allow fit parameters related

calculate only once

Add control options, base on the same structure, we can extract it automatically

$$-\ln L_1 - \ln L_1 = -\sum_{data1+data2} \ln|A|^2 + N_1 \ln \sum_{mc1} |A|^2 + N_2 \ln \sum_{mc2} |A|^2$$