

# A general and user-friendly partial wave analysis framework

## TF-PWA

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

Collaborator: Hao Cai<sup>1</sup>, Chen Chen<sup>2</sup>, Pei-Rong Li<sup>4</sup>, Yinrui Liu<sup>3</sup>, Xiao-Rui Lyu<sup>3</sup>, Rong-Gang Ping<sup>5</sup>,  
Wenbin Qian<sup>3</sup>, Mengzhen Wang<sup>2</sup>, Zi-Yi Wang<sup>3</sup>, Liming Zhang<sup>2</sup>, Yang-Heng Zheng<sup>3</sup>

<sup>1</sup>WHU, <sup>2</sup>THU, <sup>3</sup>UCAS, <sup>4</sup>LZU, <sup>5</sup>IHEP

Monday, 16 August 2021

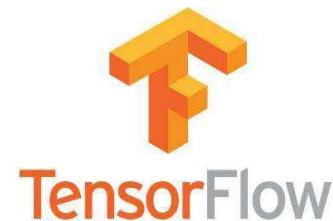
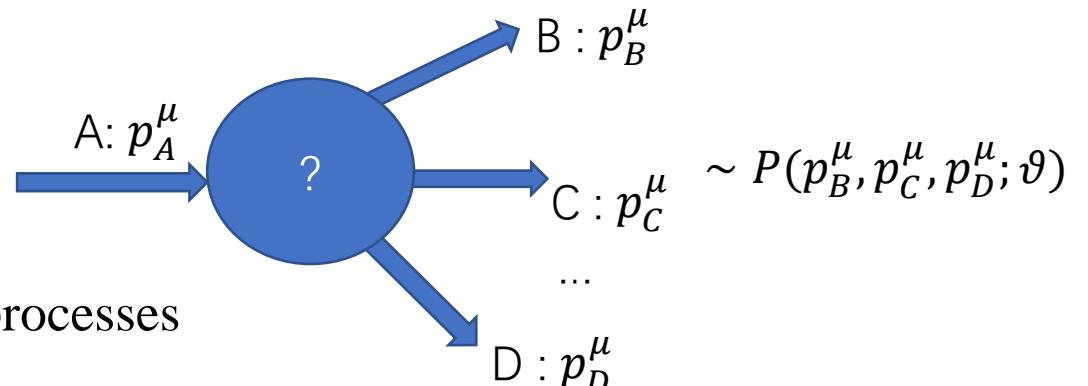
中国物理学会高能物理分会第13届全国粒子物理学学术会议

# Outline

- Introduction
- **TF-PWA**
  - Framework (aim for general propose amplitude analysis)
    - Decay process
    - Helicity amplitude
  - Special technology (to make it fast and easy to use)
    - Automatic differentiation
    - Factor system
  - Benchmarks
- Summary

# Introduction

- 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.
- We have developed a new framework using TensorFlow



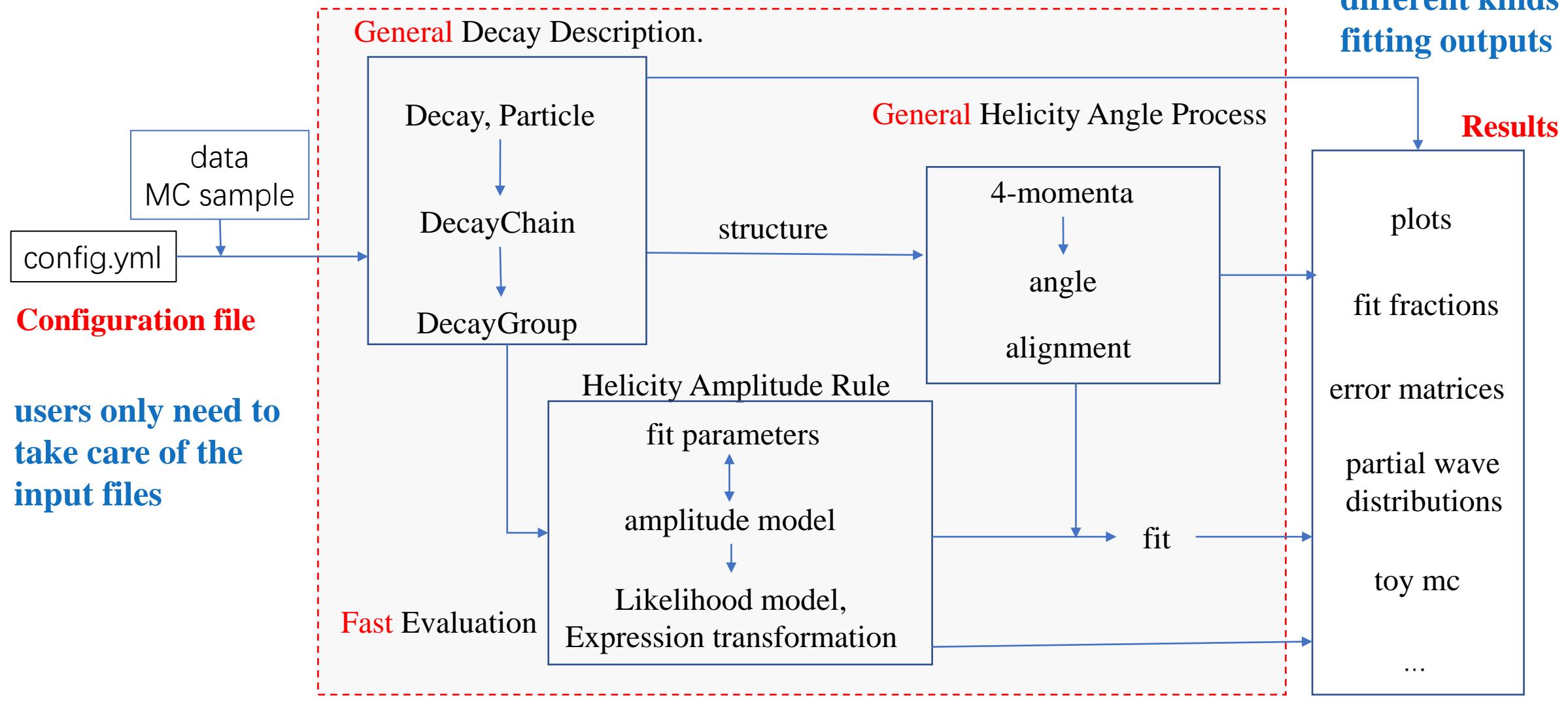
# TF-PWA: Partial Wave Analysis with TensorFlow

- Fast
  - GPU based
  - Vectorized calculation
  - Automatic differentiation
    - Quasi-Newton Method: `scipy.optimize`
  - Custom model available
- General
- Easy to use
  - Simple configuration file (example provided)
  - Most of the processing is automatic
  - All necessary functions implemented (details later)
- Open access and well supported <https://github.com/jiangyi15/tf-pwa>

# The Framework



Automatic generations of different kinds of fitting outputs



# Configuration File

- Basic Parameters

Decay:

in state, **out states**  
model (helicity amplitude  
as default)  
model parameters.

Particle:

spin parity ( $J^P$ ),  
mass ( $m_0$ )  
width ( $\Gamma_0$ ),  
model (Breit-Wigner),  
model parameters.

YAML (YAML Ain't Markup Language) format <https://yaml.org>

config.yml

```
decay:  
  B0:  
    - [DK , Dst, model:  
        default, p_break: True]  
    - [DstK , D, p_break: True]  
    - [DstD , K, p_break: True]  
  DstD: [Dst, D]  
  DstK: [Dst, K]  
  DK: [D, K]  
  Dst: [D0, pi]
```

Resonances.yml

```
Zc(3900): {J: 1, P: +1, mass:  
            3.8817, width: 0.0266, model:  
            default}  
Ds1(2700): {J: 1, P: -1,  
            mass: 2.7083, width: 0.120}  
Ds2(2573): {J: 2, P: +1,  
            mass: 2.5691, width: 0.0169}
```

Full example

- Widely used in other packages
- Easy to read and write
- Easy to rewrite in scripts

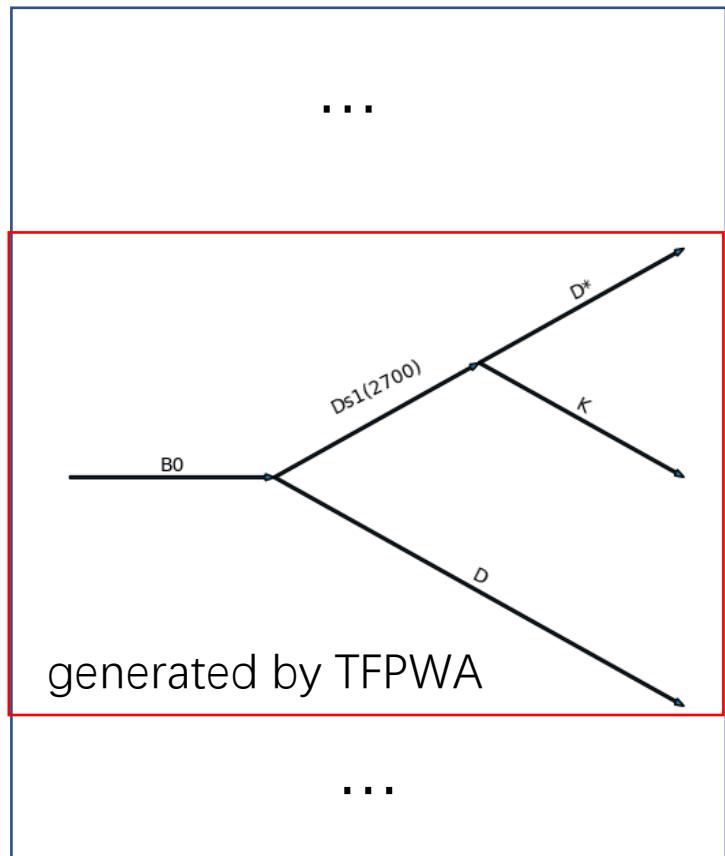
$B^0 \rightarrow D^{*-} D^0 K^+$ ,  $D^{*-} \rightarrow D^0 \pi^-$  config.yml

particle:

```
$top:  
  B0: {J: 0, P: -1, mass: 5.2}  
$finals:  
  D: {J: 0, P: -1, mass: 1.86}  
  K: {J: 0, P: -1, mass: 0.5 }  
  D0: {J: 0, P: -1, mass: 1.86}  
  pi: {J: 0, P: -1, mass: 0.14}  
DstD: [Zc(3900)]  
DstK: [Ds1(2700), Ds2(2573)]  
DK: []  
Dst: {J: 1, P: -1, mass: 2.01,  
      model: one}  
$include: Resonances.yml
```

The only places users need to modify for different decay processes

# Topology based algorithm



$B^0 \rightarrow D^{*-}D^0K^+$ : decay group

One decay chain  
 $B^0 \rightarrow D_{s1}(2700)^+ (\rightarrow D^0 K^+) D^{*-}$

dg = DecayGroup([dec1, dec2, dec3])

Automatically built from config.yml

Decays

$B^0 \rightarrow D_{s1}(2700)^+ D^{*-}$   
 $D_{s1}(2700)^+ \rightarrow D^0 K^+$   
etc.

Particles

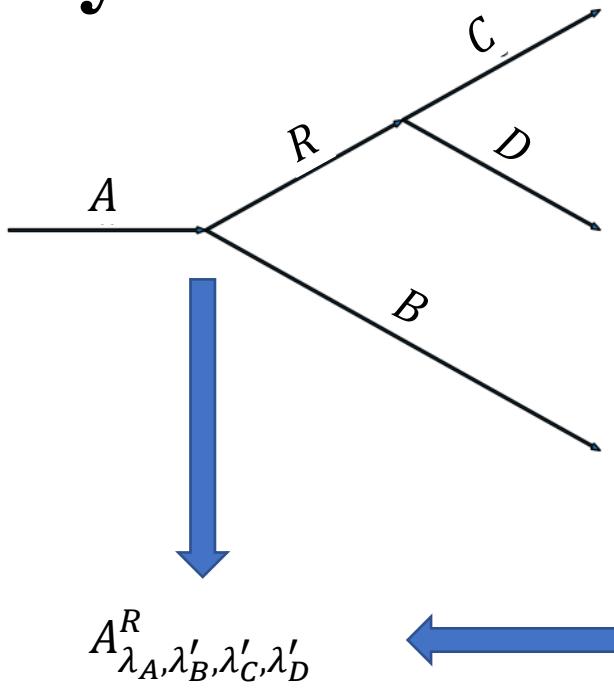
initial state ( $B^0$ ),  
final states ( $D^{*-}, D^0, K^+$ )  
propagator ( $D_{s1}(2700)$ )

Ds1 = Particle("Ds1(2700)", J=1, P=-1)  
Dst = Particle("Dst", J=1, P=-1)  
B = Particle("B", J=0, P=-1)  
...

d1 = Decay(B, [Ds1, D])

dec1 = DecayChain([d1, d2, d3])

# Helicity formalism



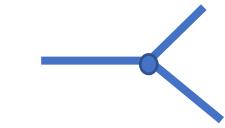
$$\sum_{\lambda} F_{\lambda_R \lambda_B} D_{\lambda_A, \lambda_R - \lambda_B}^{j_A^*}(\varphi_1, \theta_1, 0) R(M) F_{\lambda_C \lambda_D} D_{\lambda_R, \lambda_C - \lambda_D}^{j_R^*}(\varphi_2, \theta_2, 0)$$

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

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

Automatically calculated from decay structure

Feynman rules



Decay

Particle



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

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

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

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

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

User defined

$$F_{\lambda_1, \lambda_2} D_{\lambda_0, \lambda_1 - \lambda_2}^{j_0^*}(\varphi, \theta, 0)$$

Wigner-D matrix

$$R(M) = \frac{1}{m_0^2 - M^2 - im_0\Gamma}, \dots$$

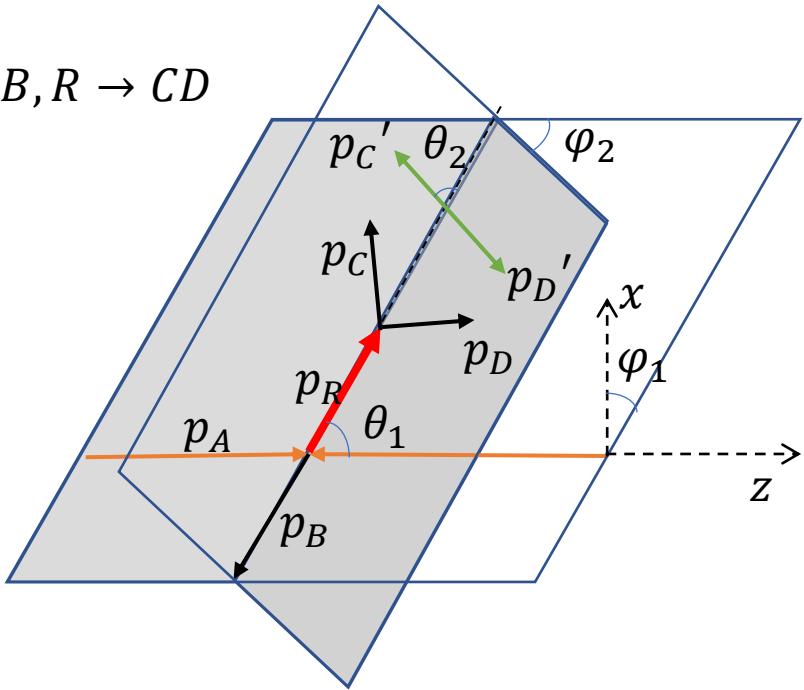
1

$$D_{\lambda_1, \lambda_1'}^{j_1^*}(\alpha, \beta, \gamma)$$

alignment

# General Helicity Angle Process

$A \rightarrow RB, R \rightarrow CD$



- Data inputs from user
  - 4-momenta ( $p^\mu$ ), weights
- Converted into amplitude variables
  - Invariant masses ( $m$ )
  - Helicity angles ( $\varphi_i, \theta_i$ )
  - Alignment angles ( $\alpha, \beta, \gamma$ )

Automatically calculated based on decay structure,  
using Euler rotation system, boost sequence, and SU(2) relationship.

[M. Wang, Y. Jiang, Y. Liu et.al., Chinese Physics C Vol. 45, No. 6 \(2021\) 063103](#)

# Automatic Differentiation (AD)

- AD provides a powerful way to calculate gradient automatically

- Time for calculating gradient  
 $N$  is the number of parameters  
 $t$  is the time of function evaluation

AD	$[f(x + \delta) - f(x)]/\delta$
$\sim t$	$\sim N t$

supported fit method

- Fast Fit

- can reduce time for many gradient based method

$$f(x) \approx f(x_0) + \frac{\partial f}{\partial x}(x - x_0) + \frac{\partial^2 f}{\partial x_i \partial x_j} (x - x_0)_i (x - x_0)_j \dots$$

$$x_{out} = \operatorname{argmin} f(x) \Rightarrow x_{out} \approx \left( \frac{\partial^2 f}{\partial x_i \partial x_j} \right)^{-1} \frac{\partial f}{\partial x}$$

- Error propagation

$$\sigma_y = \sqrt{\frac{\partial y}{\partial x_i} V_{ij} \frac{\partial y}{\partial x_j}}$$

- Calculate uncertainties of physical observables , (such as fit fractions)
- Easy to use in TFPWA

Fit method	type
Nelder-Mead	no-gradient
BFGS	gradient
CG	gradient
Newton-CG	hessian,
trust-krylov	hessian vector product
trust-ncg	
...	
iminuit	gradient

scipy

can calculated by AD

# Factor system: automatic factorization of amplitude

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

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

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

- Some special treatment is implemented

- Plotting of partial waves ( $P_i = |A_i|^2$ )

- Amplitude caching

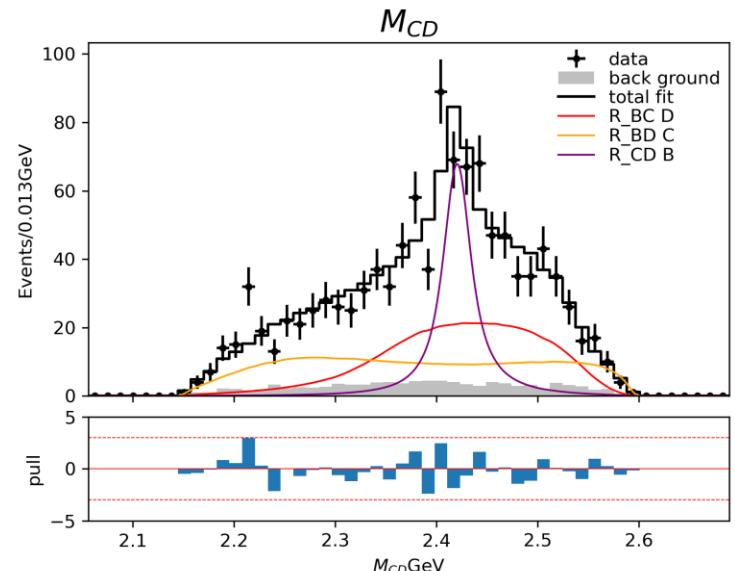
- mass dependent:

fit parameters related

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

- factor only:  $\sum |A|^2 \rightarrow G_i G_j^* (\sum C_{ij})$

calculate only once



method	Time for Calculating -ln L (s)	Time for Single fit (s)	Memory (MB)
original	1.00243	224.1	4371
mass dependent	0.41840	103.2	2323
factor only	0.17831	31.56	2323

# Benchmarks

CPU: i7-9750H@2.6 Hz with 12 cores

GPU: Nvidia 1660 Ti, a cheap GPU

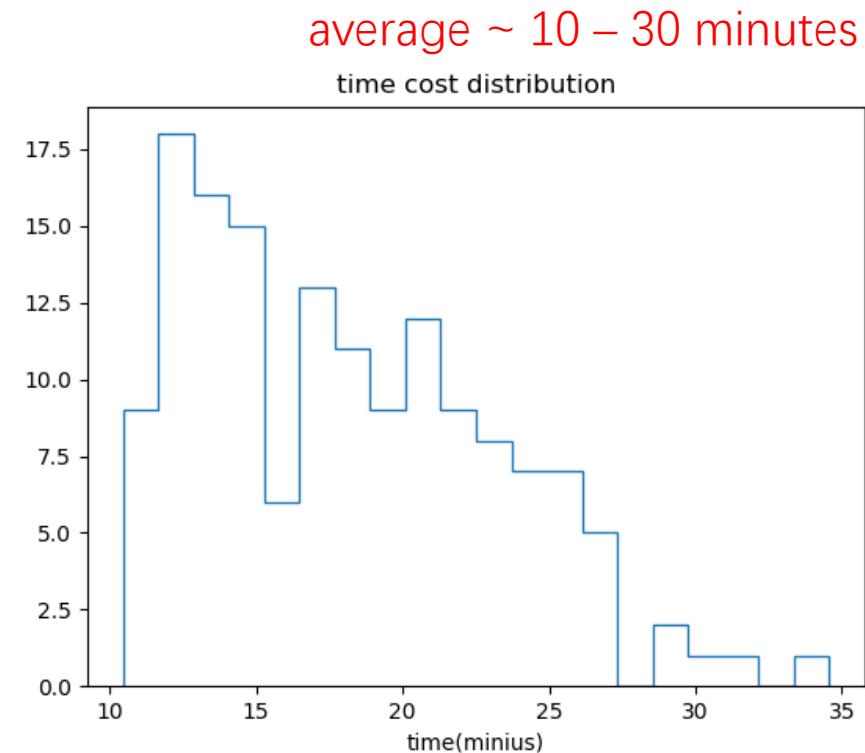
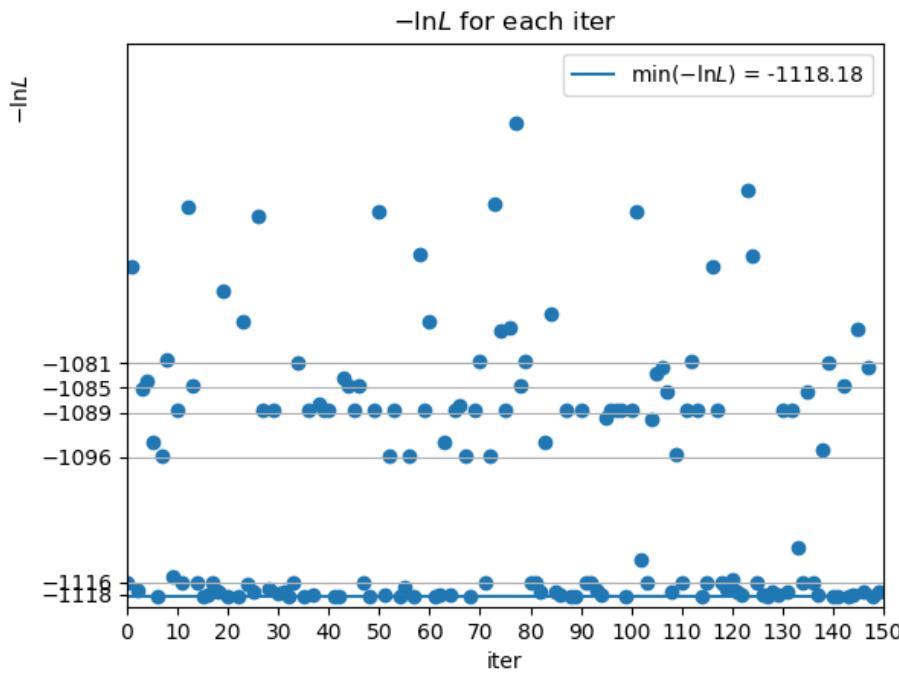
Test based on simple MC (200000) sample of a simple amplitude model of  
 $e^+e^- \rightarrow R_1(1^+)\pi, R_1 \rightarrow D^*D^*$

Both based on TF-PWA

process	CPU (ms)	GPU (ms)	Ratio
Sum of log(PDF)	725 <b>X 12</b>	91	$\sim 8 \text{ X } 12 = 96$
Sum + differential	1417 <b>X 12</b>	188	$\sim 7.5 \text{ X } 12 = 90$

Almost a factor of **100 times** faster for GPU than single CPU

# Performance in a real fit



- Randomizing of initial parameters, better searching for global minimum
- Toy studies
- Uncertainty studies etc.

# Functions implemented

- Toy studies
- Plotting
- Fit fractions, interference fractions
- Simultaneous fit between different datasets
- Parity conversation, isospin constraints
- Gaussian constraints on parameters
- 2D chi2 test
- CP violation fit
- Final states with identical particles
- Amplitude factorization
- Resolution
- Simple symbolic formula
- Error propagation
- ...

# Summary

- A general framework is developed for PWA based on TensorFlow
- Fast, easy to use and still in active development
- TF-PWA cross-checked with other fitters used in BESIII and LHCb collaborations
- Have been used for a large variety of amplitude analyses
- Well supported , welcome to use and provide feedback.

Thank you for your attentions!

# Back Up

# Fit process: maximum likelihood

$$P(x_i; \vartheta) = \frac{1}{\sigma_{tot}} \frac{d\sigma}{d\Phi}(x_i; \vartheta), \sigma_{tot} = \sum_{x_i \in MC} \omega_i \frac{d\sigma}{d\Phi}(x_i; \vartheta), \omega_i = \frac{1}{N_{MC}}$$

Negative Log-Likelihood (NLL):

$$-\ln L = -\alpha \sum_{x_i \in \text{data}} w_i \ln P(x_i; \vartheta), \alpha = \frac{\sum w_i}{\sum w_i^2}$$

Can either subtract background using weights or directly include background in the fit

$$\hat{\vartheta} = \operatorname{argmin} (-\ln L)$$

BFGS:

$$\begin{aligned}\vartheta_{i+1} &= \vartheta_i + kp_i \\ p_i &= -B_i g_i \\ B_{i+1} &= B_i + \frac{y_i y_i^T}{y_i^T s_i} - \frac{B_i s_i s_i^T B_i}{s_i^T B_i s_i} \\ s_i &= \vartheta_{i+1} - \vartheta_i, y_k = g_{i+1} - g_i\end{aligned}$$

$$g_i = -\frac{\partial \ln L}{\partial \vartheta}$$

Automatic differentiation:

- Reduce the number of calculation of  $-\ln L$ , speed up the fit.
- More accurate gradient, improve the fit result

error matrix:

$$V_{ij}^{-1} = -\frac{\partial \ln L}{\partial \vartheta_i \partial \vartheta_j} |_{\vartheta=\hat{\vartheta}}$$

also calculated by automatic differentiation

$$V \approx B$$

# Custom Model

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

```
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 """
        m = args[0]["m"]
        zeros = tf.zeros_like(m)
        return tf.complex(m + self.a(), zeros)
```

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

$$F_{[\lambda_R \lambda_B]}(x; \vartheta) D_{[\lambda_A, \lambda_R - \lambda_B]}^{j_A \star}(x) R(x; \vartheta) F_{[\lambda_C \lambda_D]}(x; \vartheta) D_{[\lambda_R, \lambda_C - \lambda_D]}^{j_R \star}(x)$$
$$D_{[\lambda_B, \lambda'_B]}^{j_B \star}(x) D_{[\lambda_C, \lambda'_C]}^{j_C \star}(x) D_{[\lambda_D, \lambda'_D]}^{j_D \star}(x) \rightarrow A_{[\lambda_A, \lambda'_B, \lambda'_C, \lambda'_D]}(x; \vartheta)$$
$$\lambda_{[RB][ARB][CD][RCD][BB'][cc'][DD']} \rightarrow [AB'C'D']$$

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

$R(x; \vartheta)$

$x$ : all data, (\*args, \*\*kwargs)

$\vartheta$ : 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`.

# Resonant models and LS coupling

## Default Model:

- $R(M)$ : Breit-Wigner with running width

$$R(M) = \frac{1}{M^2 - m_0^2 + m_0 \Gamma(M)}, \quad \Gamma(M) = \Gamma_0 \left( \frac{q}{q_0} \right)^{2l_0+1} \frac{m_0}{m} B_{l_0}^{'2}(q, q_0, d)$$

- LS coupling: from helicity base to LS base

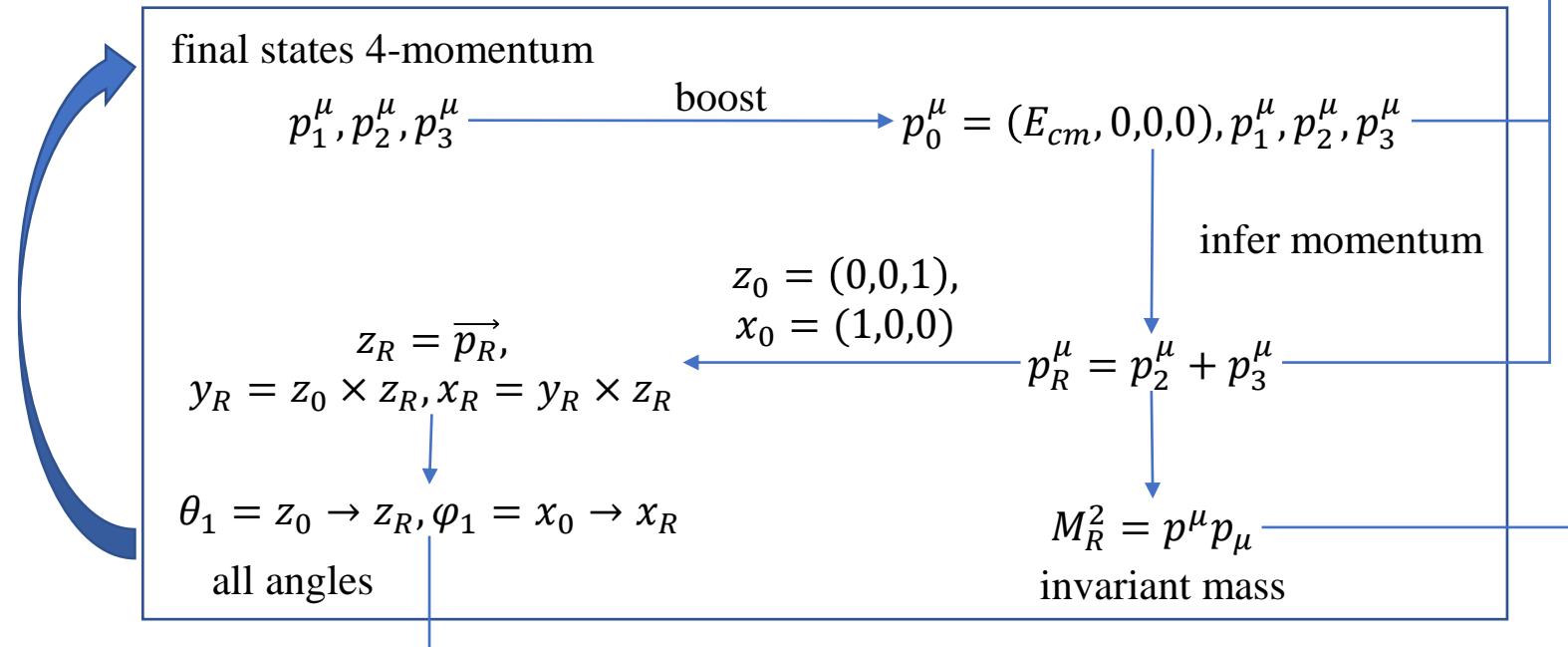
- $F_{\lambda_1, \lambda_2} : J^P(\lambda_0) \rightarrow j_1^{\eta_1}(\lambda_1) + j_2^{\eta_2}(\lambda_2)$

LS coupling:  $|j_1 - j_2| < s < j_1 + j_2$ ,  $|l - s| \leq J \leq l + s$

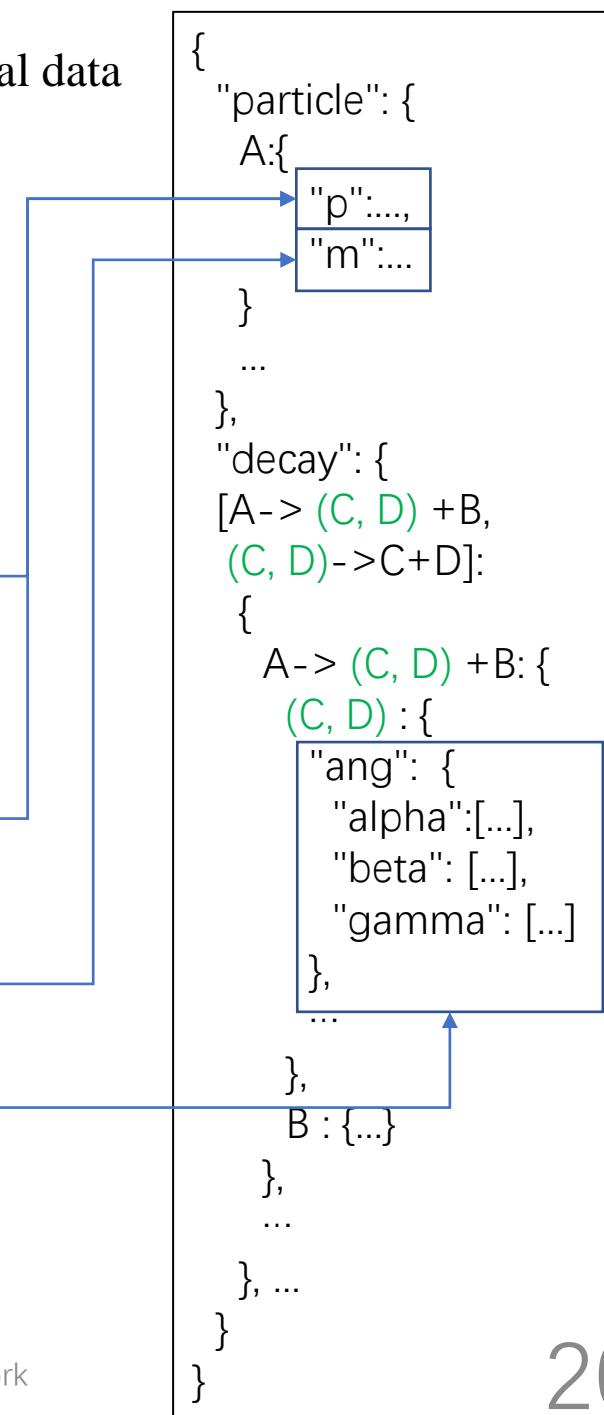
Parity conservation:  $P = \eta_1 \eta_2 (-1)^l$

# Data process: Automatic angle calculation

$$p_0 \rightarrow p_1 + p_2 + p_3: \quad p_0 \rightarrow p_1 + p_R, p_R \rightarrow p_2 + p_3$$



Total data



loop for decay and decay chain,  
so that all angles are calculated automatically

# Alignment angle

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  $0 \rightarrow R, 2; R \rightarrow 1, 3$ :

$$|p_1\rangle_R = B_z(\omega_1)R_z(0)R_y(\theta_1)R_z(\phi_1)|p_1\rangle = L_1|p_1\rangle$$

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

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

$$|p_1\rangle_{R'} = B_z(\omega'_1)R_z(0)R_y(\theta'_1)R_z(\phi'_1)|p_1\rangle = L'_1|p_1\rangle$$

$$|p_1\rangle_2 = B_z(\omega'_2)R_z(0)R_y(\theta'_2)R_z(\phi'_2)|p_1\rangle = L'_2|p_1\rangle_{R'}$$

The alignment angle is the rotation

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

So

$$L_r = B_z(\omega)R_z(\gamma)R_y(\beta)R_z(\alpha) = L'_2 L'_1 L_1^{-1} L_2^{-1}$$

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

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

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

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

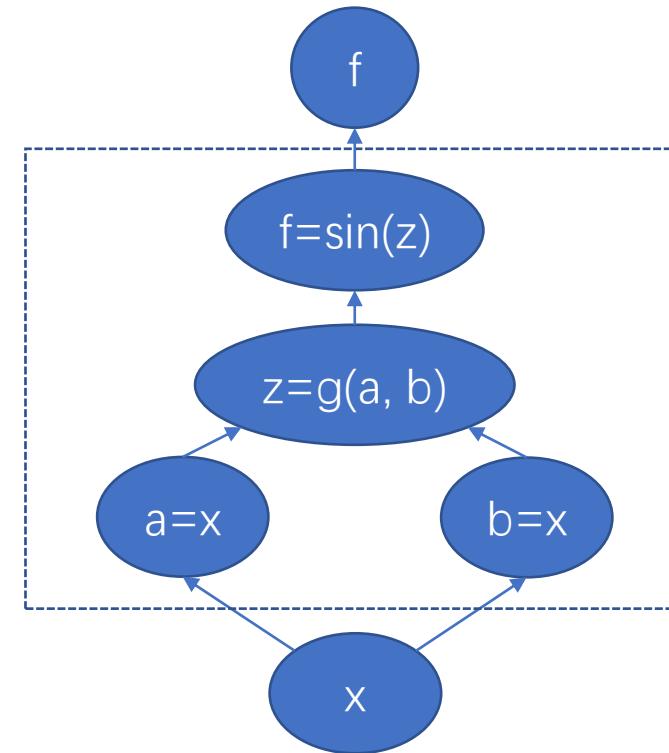
$$\omega = \ln \left| \frac{L_{22}}{L_{11}} \right|$$

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

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

# Automatic Differentiation (AD)

- AD provides a powerful way to calculate gradient automatically
- Chain Rules
  - $\times$ :  $\frac{\partial f(g(x))}{\partial x_i} = \frac{df}{dg} \times \frac{\partial g}{\partial x_i}$
  - $+$ :  $\frac{\partial f(h(x), g(x))}{\partial x_i} = \frac{\partial f}{\partial h} \frac{\partial h}{\partial x_i} + \frac{\partial f}{\partial g} \frac{\partial g}{\partial x_i}$



Only basic operator need for differentiation.  
Similar as amplitude rules

Automatic Differentiation (numerically):

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

$\xleftarrow{\quad}$  forward  
 $\xrightarrow{\quad}$  backward

$$f = \sin(x^x) = \sin(z), z = x^x$$

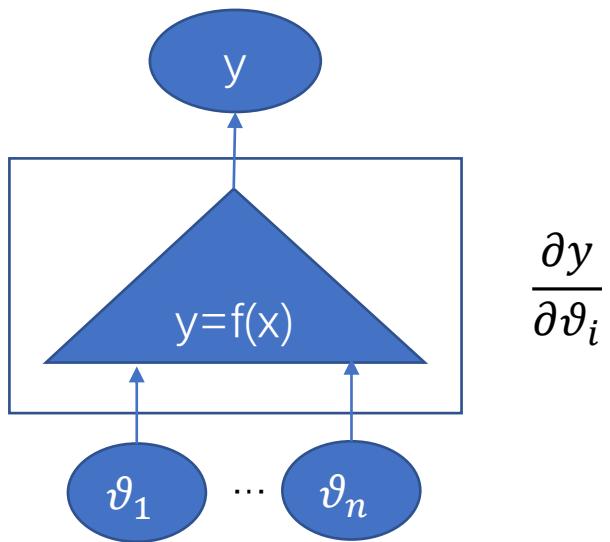
$$g(a, b) = a^b$$

$$\frac{df}{dz} = \cos z, \frac{\partial g(a, b)}{\partial a} = ba^{b-1}, \frac{\partial g(a, b)}{\partial b} = a^b \ln a$$

Symbolic method:

$$\begin{aligned} \frac{df}{dx} &= \frac{df}{dz} \frac{dg(x, x)}{dx} = \cos z \left[ (ba^{b-1}) \Big|_{a,b=x} + (a^b \ln a) \Big|_{a,b=x} \right] \\ &= \cos x^x [xx^{x-1} + x^x \ln x] = x^x \cos x^x (\ln x + 1) \end{aligned}$$

# Reduce Memory Usage



n: number of parameters  
m: number of data  
s: model complexity

AD time and space complexity

Complexity	time	memory
forward	$O(mns)$	$O(mn)$
backward	$O(ms)$	$O(ms)$

memory is dependent on number of data for each differentiation calculation

- split MC into batches  $MC \rightarrow \{MC_0, MC_2, \dots\} \Rightarrow memory \downarrow$

$$-\frac{\partial \ln L}{\partial \vartheta} = -\alpha \frac{\partial \left[ \sum w_i \ln \frac{d\sigma}{d\Phi}(x_i) \right]}{\partial \vartheta} + \frac{\alpha \sum w_i}{\sum_j I_j} \left( \sum_j \frac{\partial I_j}{\partial \vartheta} \right), \quad I_j = \sum_{i \in MC_j} \omega_i \frac{d\sigma}{d\Phi}(x_i; \vartheta)$$

# AD transform

All parts can be calculated from small batches of data.

- $p(x; \vartheta) = \frac{f(x; \vartheta)}{N_f(y; \vartheta)} + g(x); N_f(x) = \sum_{i \in MC} f(x_i; \vartheta)$
- $\frac{\partial p(x; \vartheta)}{\partial \vartheta} = \frac{\partial p(x; \vartheta; N_f)}{\partial \vartheta} + \frac{\partial p(x; \vartheta; N_f)}{\partial N_f} \frac{\partial N_f}{\partial \vartheta}$
- $\frac{\partial N_f}{\partial \vartheta} = \sum_{i \in MC} \frac{\partial f(x_i; \vartheta)}{\partial \vartheta} = \sum_i \sum_{j \in MC_i} \frac{\partial f(x_i; \vartheta)}{\partial \vartheta}$
- $-\frac{\partial \ln L}{\partial \vartheta} = -\alpha \frac{\partial \sum w_i \ln p(x; \vartheta; N_f)}{\partial \vartheta} - \alpha \frac{\partial \sum w_i \ln p(x; \vartheta; N_f)}{\partial N_f} \frac{\partial N_f}{\partial \vartheta}$

```
with config.params_trans() as f:  
    real = f["A->R_BD.C_g_ls_1r"]  
    imag = f["A->R_BD.C_g_ls_1i"]  
    alpha = real / (1 + real * real + imag * imag)  
    print(alpha, f.get_error(alpha))
```

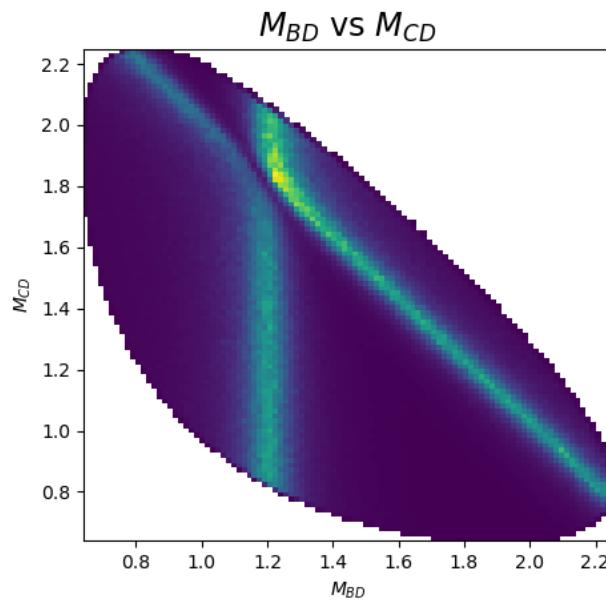
# identical particles

$$\begin{aligned} A &\rightarrow BCD \\ A &\rightarrow (BC)D, A \rightarrow (BD)C \end{aligned}$$

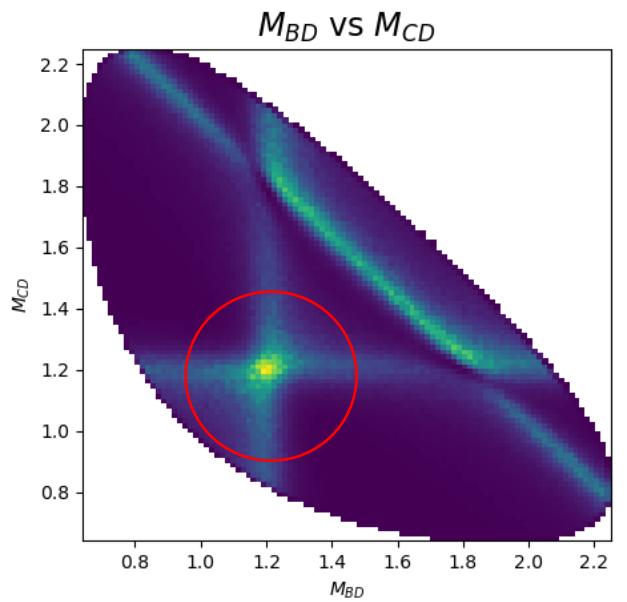
B, C are identical particles

$$Amp(p_B, p_C, p_D) \Rightarrow Amp(p_B, p_C, p_D) \pm Amp(p_C, p_B, p_D)$$

No identical particles



boson :  
Interference is enhanced



fermion:  
Interference is eliminated

