中国科学院大学
University of Chinese Academy of Sciences

# TF-PWA: a general partial wave analysis framework using TensorFlow

Yi Jiang (蒋艺)

University of Chinese Academy of Sciences

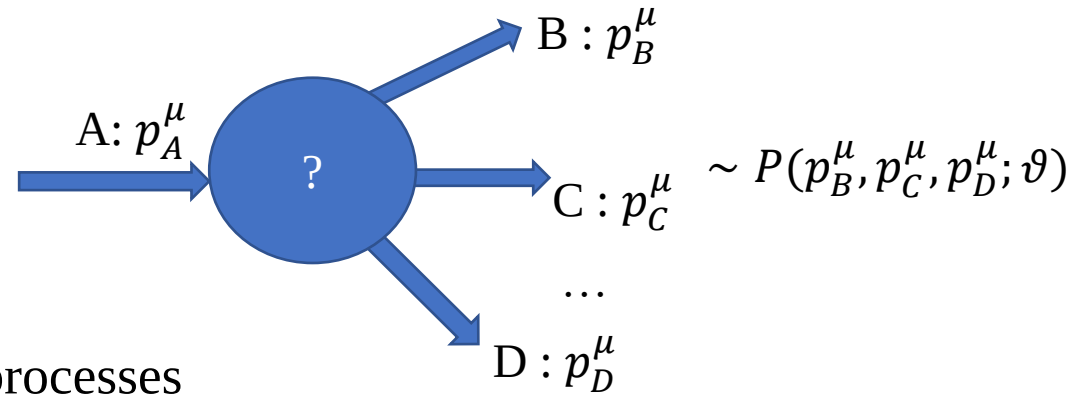
Wednesday April 16, 2025

# Outline

- Introduction
- TF-PWA
  - General designs
  - Automatic Differentiation
  - Performances of TF-PWA
- Some analysis using TF-PWA
- Prospect and 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



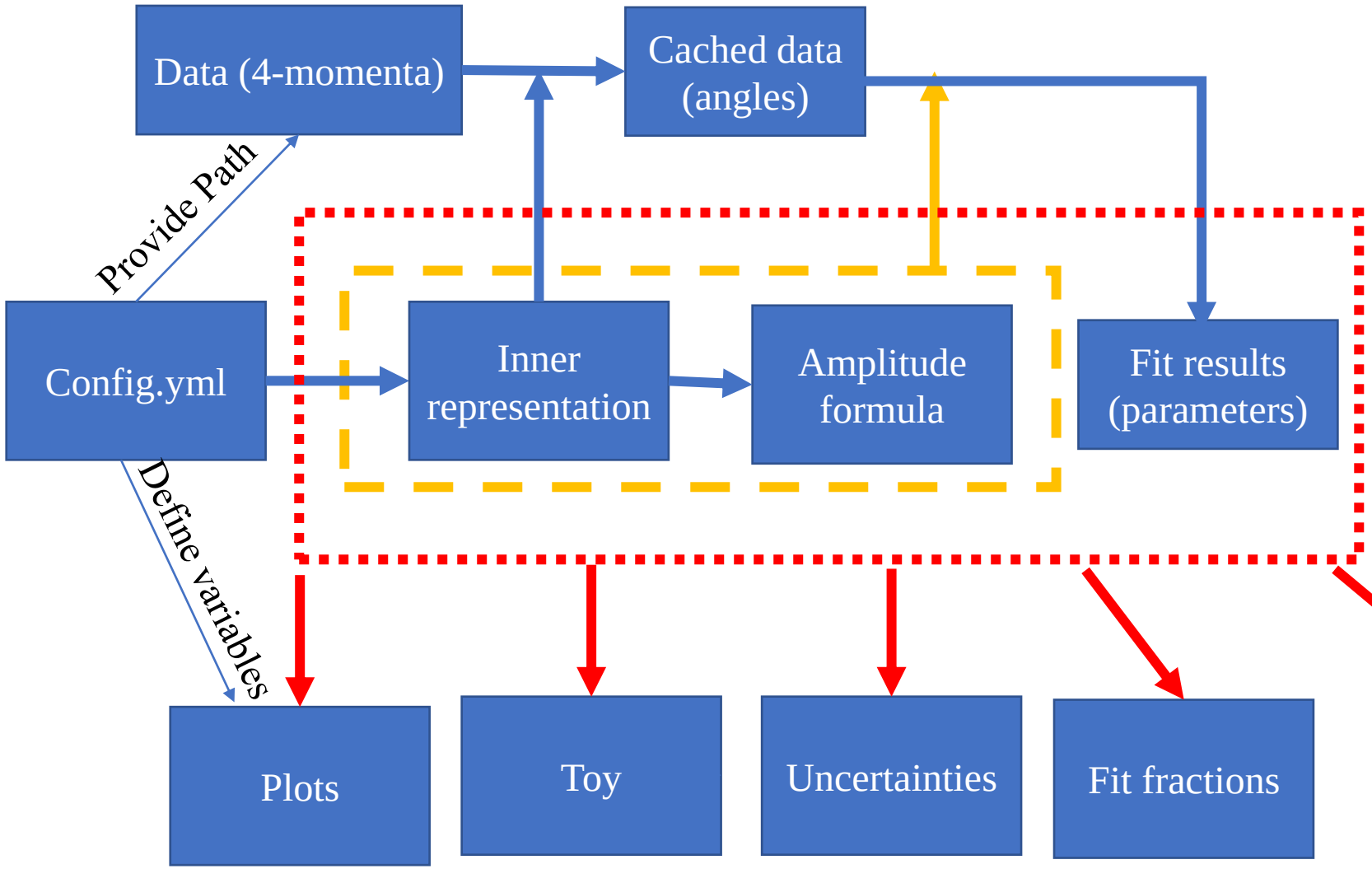

- Most of previous fitters are designed for special processes or are time-consuming.

- 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




Basic scripts

```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)

...
```

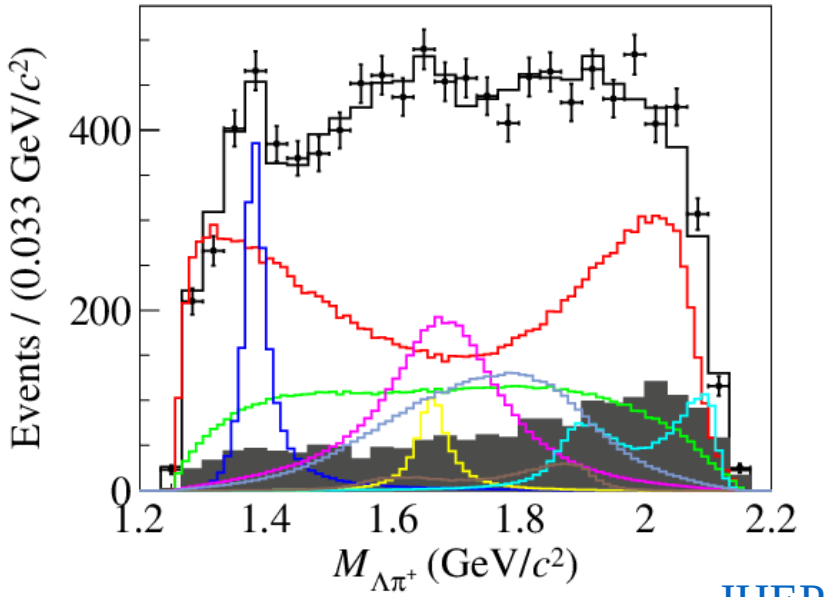
# Base Example: $\Lambda_c^+ \to \Lambda\pi^+\pi^0$

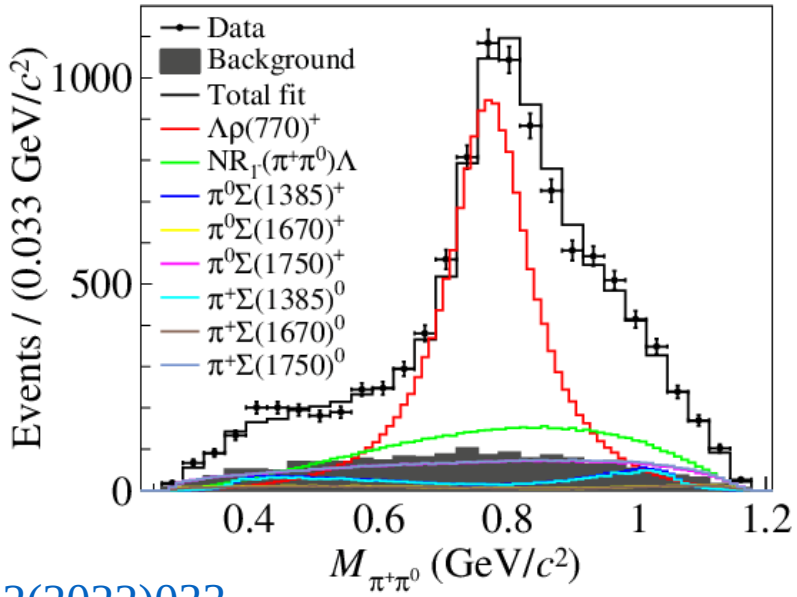
The first PWA using TF-PWA
Code: https://github.com/jiangyi15/tf-pwa-example/tree/main/lmd_pipi

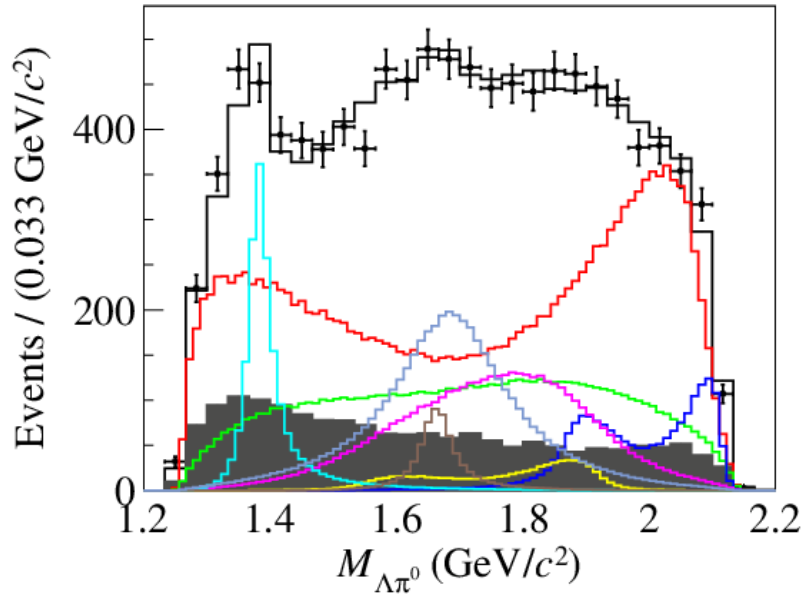
- $\Lambda_c^+ \to \Lambda(\to p\pi^-)\pi^+\pi^0$
  - Simultaneous fit
    - 7 energy points
    - Total around 10k events, 854k MC
    - 38 free parameters
  - Dominated by $\Lambda_c^+ \to \Lambda\rho$: 57.2 ± 4.2%
  - Clear peak for $\Lambda_c^+ \to \pi\Sigma(1385)$

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
```



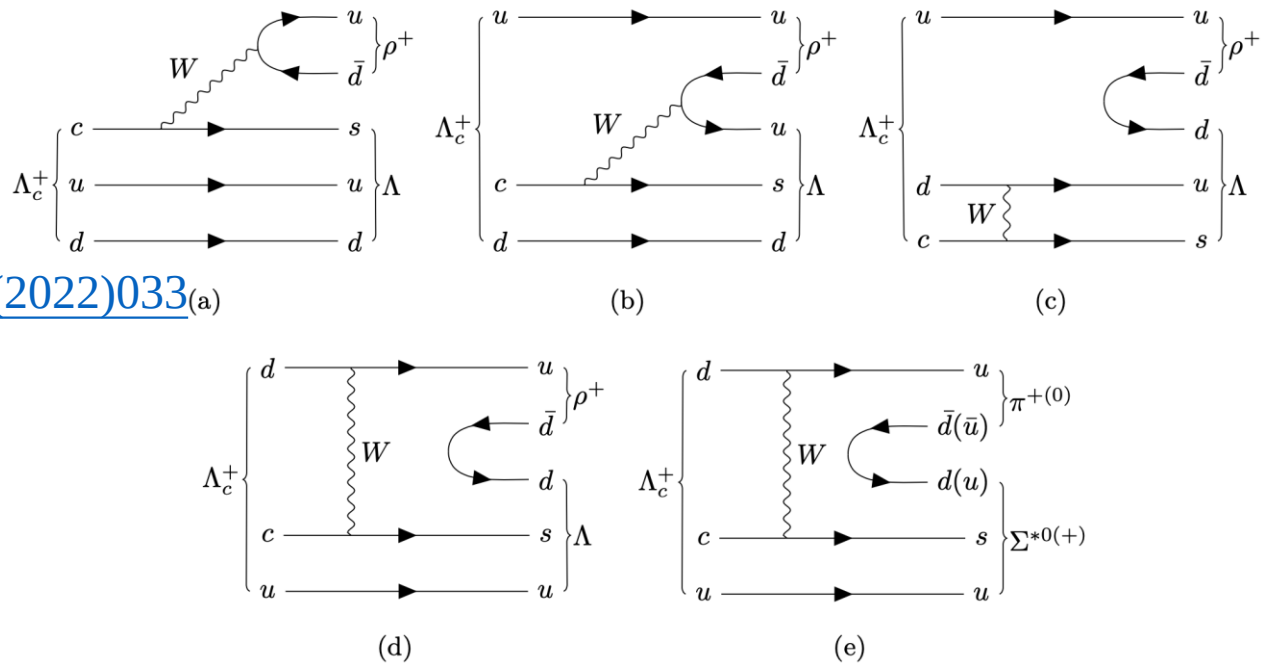




JHEP12(2022)033

# Configuration

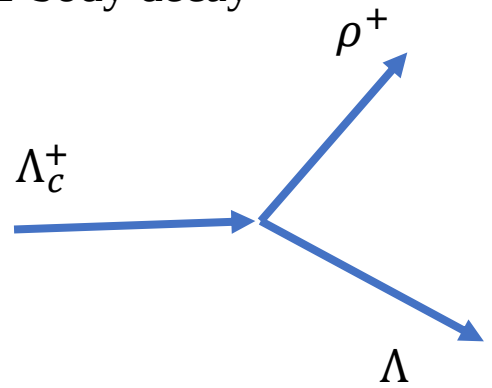
- What is needed?
  - Particles (Resonances, line) and their properties
  - Decays (interaction, vertex) and their properties
- Store in dict or list, save as YAML file.
- Possible process in this $\Lambda_c$ decay $\Lambda_c^+ \to \Lambda\pi^+\pi^0$




JHEP12(2022)033

YAML: https://yaml.org

2-body decay



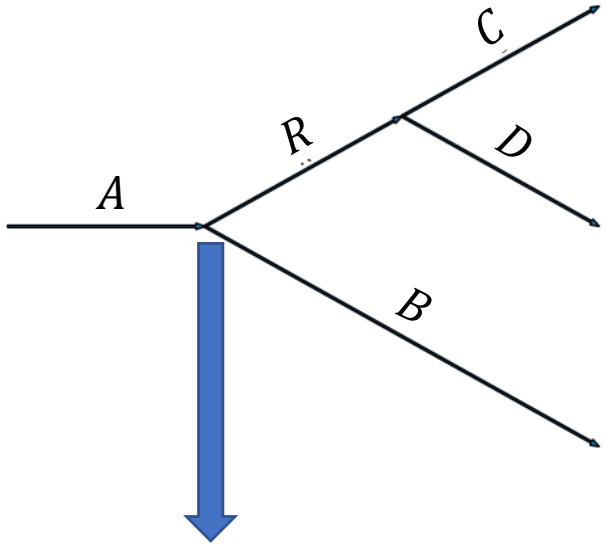

Serialize as

```
Lambda_c: [
  [Lambda, rho],
  [Sigma_starp, pi0 ],
  [Sigma_star0, pip ],
]
```

```
rho:
  J: 1
  P: -1
  mass:  0.77511
  width:  0.1491
  model: GS_rho
…
```

Config.yml In TF-PWA

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

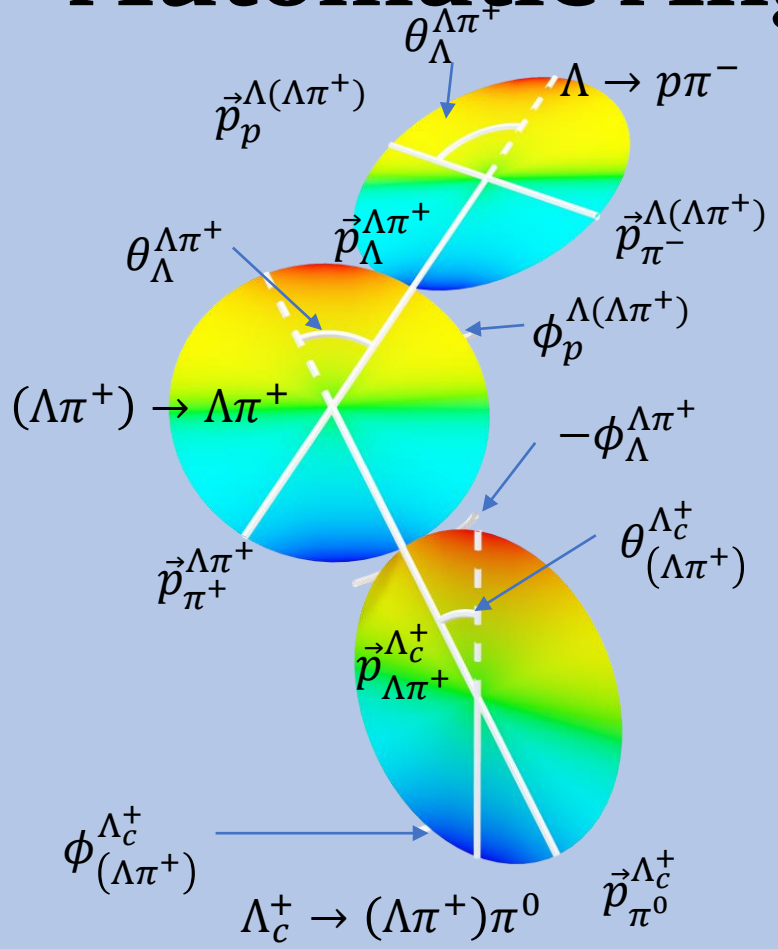
Decay Chain: $\tilde{A} = A_1 R A_2 \cdots$

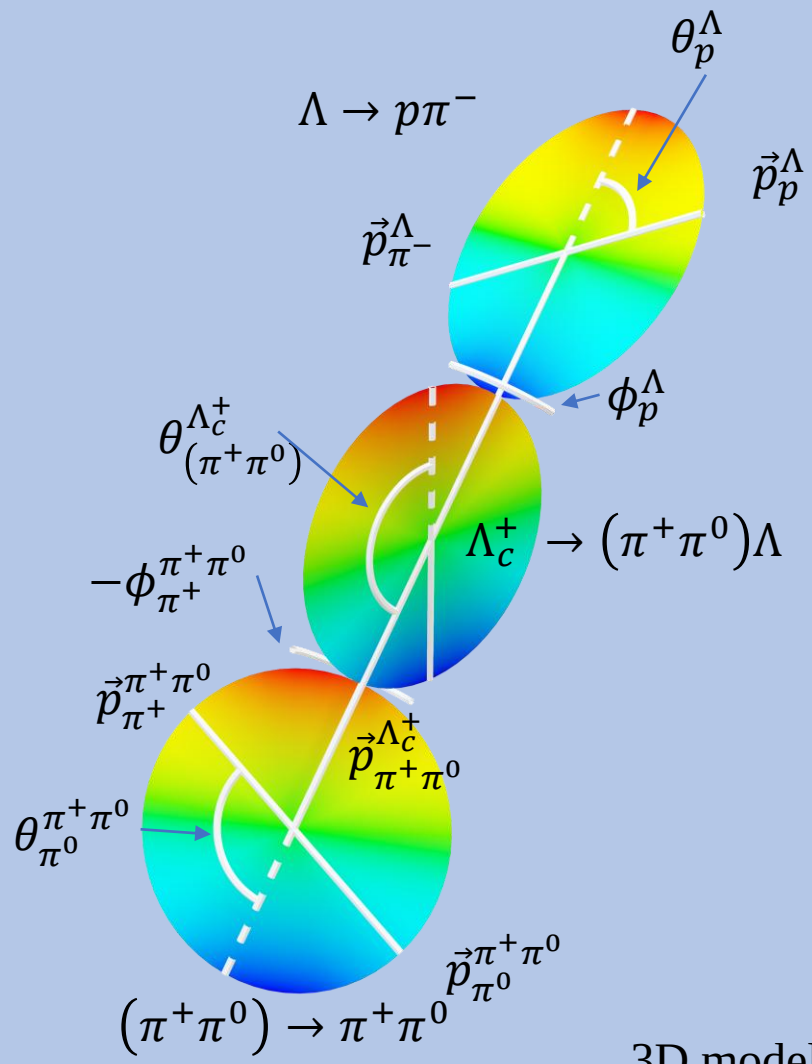
Decay: Wigner D-matrix, $A = H D^{*J}(\phi,\theta,0)$

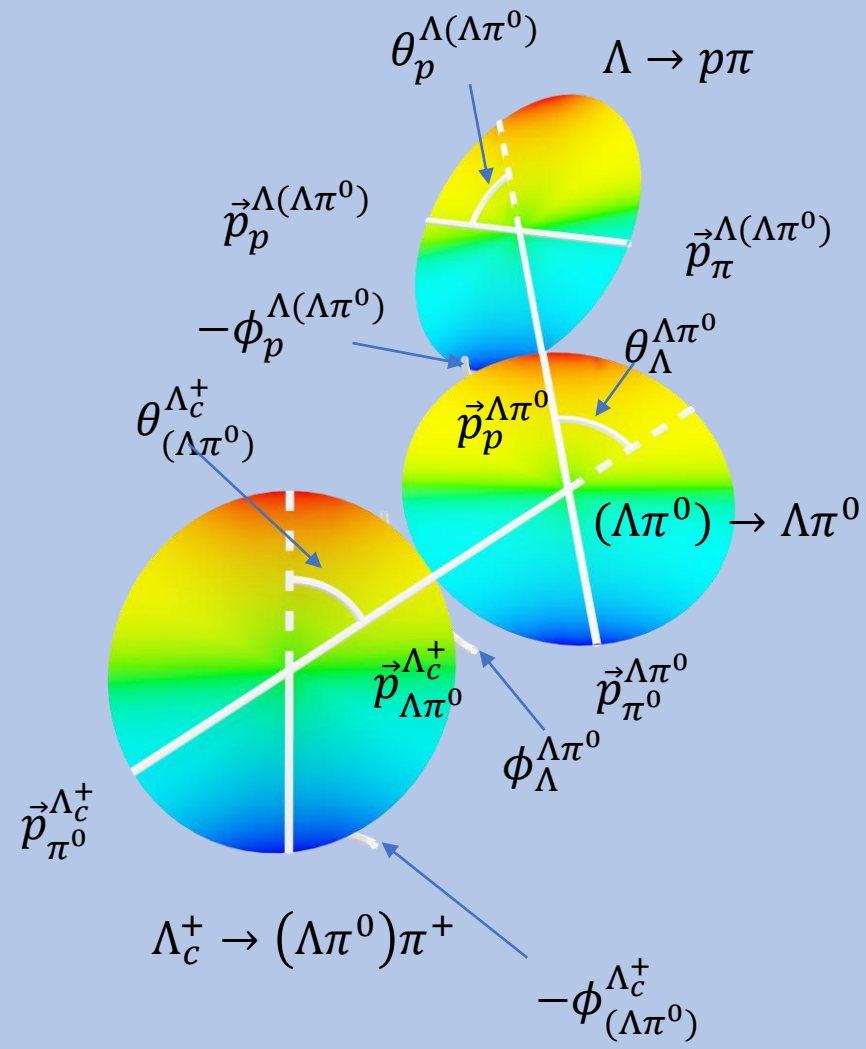
Particle: Breit-Wigner: $R(m)$, user defined

# Automatic Angle Plot

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



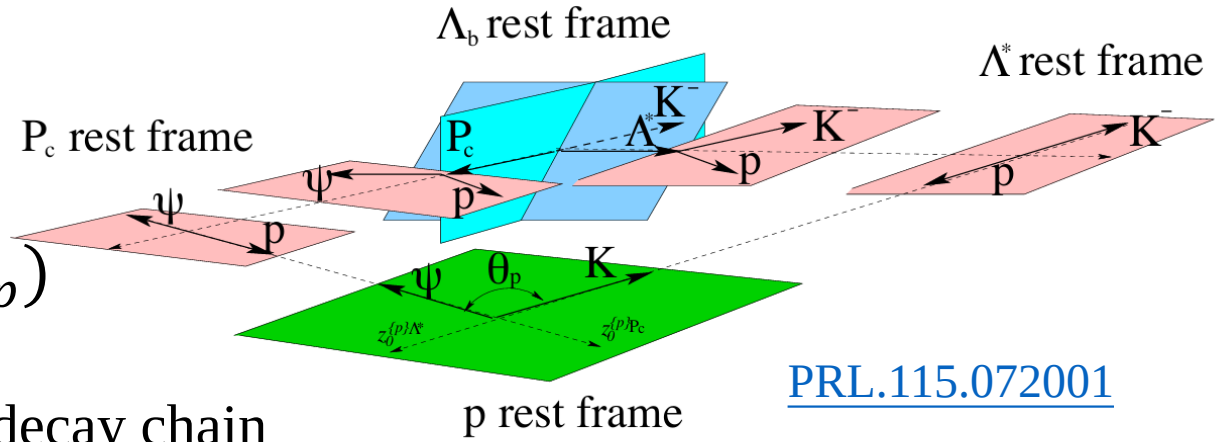




Auto calculated by TF-PWA,
Only required the 4-monmenta

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

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

# Alignment




PRL.115.072001

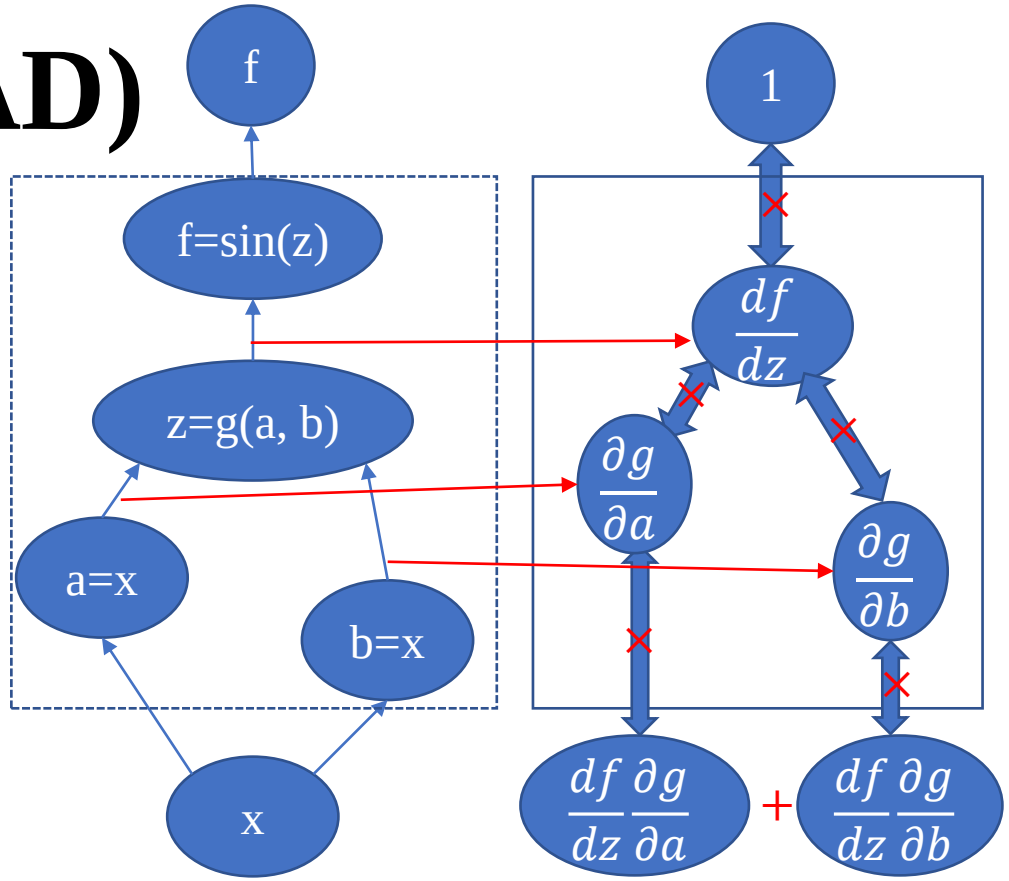
Figure 18: Definition of the $\theta_p$ angle.

- Problem found in pentaquark study ($\theta_p$)
- Helicity formula
  - Required addition alignment of different decay chain
- General formula of alignment
  - The rotation part: $R = \Lambda(Lp)^{-1}L\Lambda(p)$, for a general Lorentz transformation ($L$)
  - $\Lambda(p)$ is a pure boost from a reference momentum to momentum $p$
  - R keeps the same reference momentum: $\mathrm{R}p_0 = p_0$,
    - $p_0 = (m_0, 0,0,0),\ m_0 \neq 0$
    - $p_0 = (1,0,0,1)$ for massless particle
  - $R = \Lambda(L_2p)^{-1}L_2\Lambda(L_1p)\Lambda(L_1p)^{-1}L_1\Lambda(p) = \Lambda(L_2p)^{-1}L_2L_1\Lambda(p)$
    Irreducible tensor formula, helicity formula
  - In helicity formula
    - $p$, L$p$ are all in z-axis (as the definition of helicity), so $\Lambda(Lp)^{-1}, \Lambda(p)$ is $B_z$
    - $B_z$ does not affect the angle calculation.
  - TF-PWA provides additional option for alignment:
    - align_ref: center_mass: align to spin projection in z-axis instead of helicity.

# 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 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)
  - $\ln L(\vartheta) \Rightarrow \ln L(\vartheta; I_i)$
  - $\frac{\partial \ln L(\vartheta)}{\partial \vartheta} \Rightarrow \frac{\partial \ln L(\vartheta;I_i)}{\partial \vartheta} + \sum_i \frac{\partial \ln L(\vartheta;I_i)}{\partial I_i} \frac{\partial I_i}{\partial \vartheta}$
  - $\frac{\partial I_i}{\partial \vartheta} = \frac{\partial [\sum \omega_j P_i(x_j)]}{\partial \vartheta} + \frac{\partial [\sum \omega_j P_i(x_j)]}{\partial \vartheta} + \cdots$

    Batch 1 | Batch 2

    clear intermediate variables
- Expand the power of AD to multi GPU, even multi cluster

# 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)
    - $-\ln L = -\ln L(\vartheta, z)$, $\vartheta$: 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)))
```

# Different preprocess

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

- Implement  different ways for amplitude
  - Option in config.yml: preprocessor and  amp_model
  - Balance performance and memory usage.
- Performance of data:MC = 1:10
  - Linear performance of data size
  - Boundary of memory size

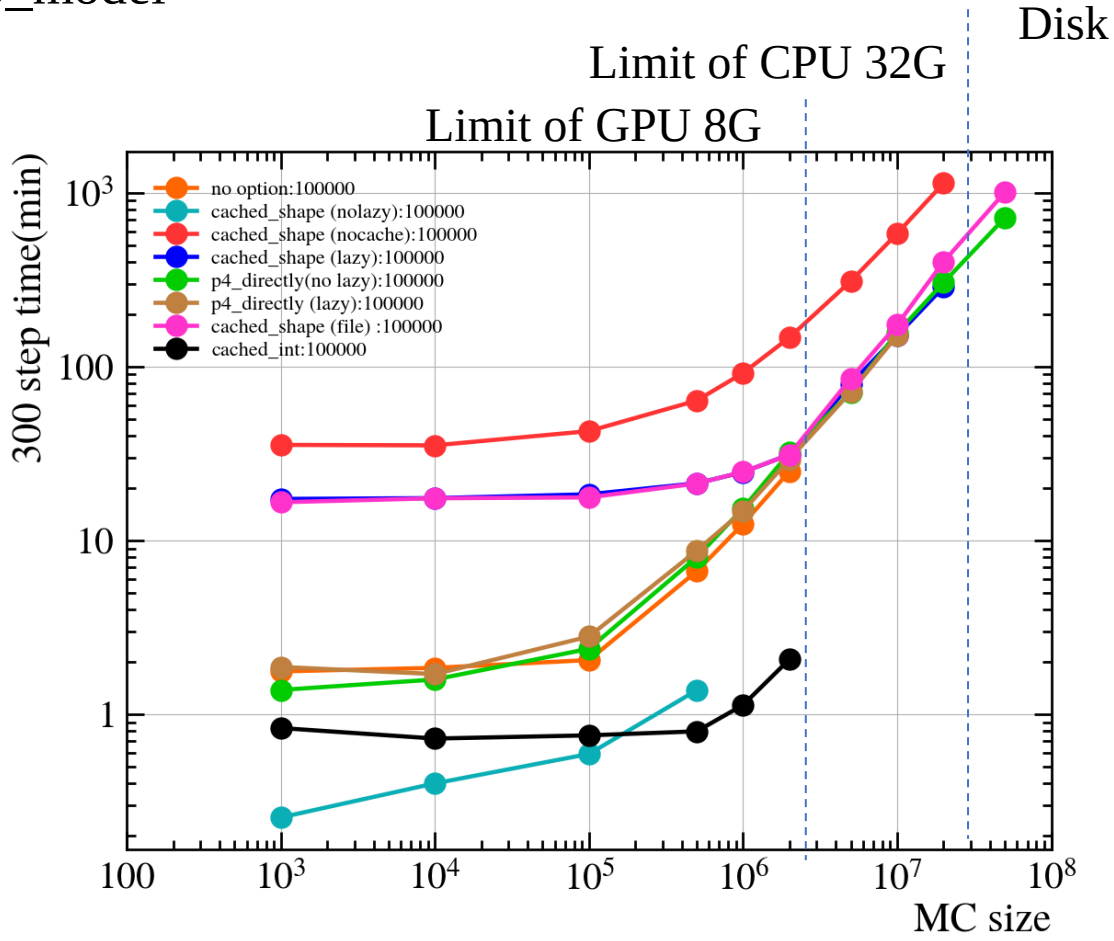
Store data in memory ⟷ Calculation in GPU

| 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 | $p^\mu \to m, angle, T_i$<br>$f_j T_j$ | $c_i f_i(m) T_i$<br>$c_j f_j T_j$ |
| p4_directly | $p^\mu$ | $p^\mu \to m, angle$<br>$c_i f_i(m) T_i(angle)$ |

Table 1: Calculation in the two parts

| data | memory requirement for one event |
|---|---|
| $p^\mu$ | 4 N(particles) |
| $m$ | N(particles) |
| $angle$ | 6 N(chain) N(decay in one chain)<br>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

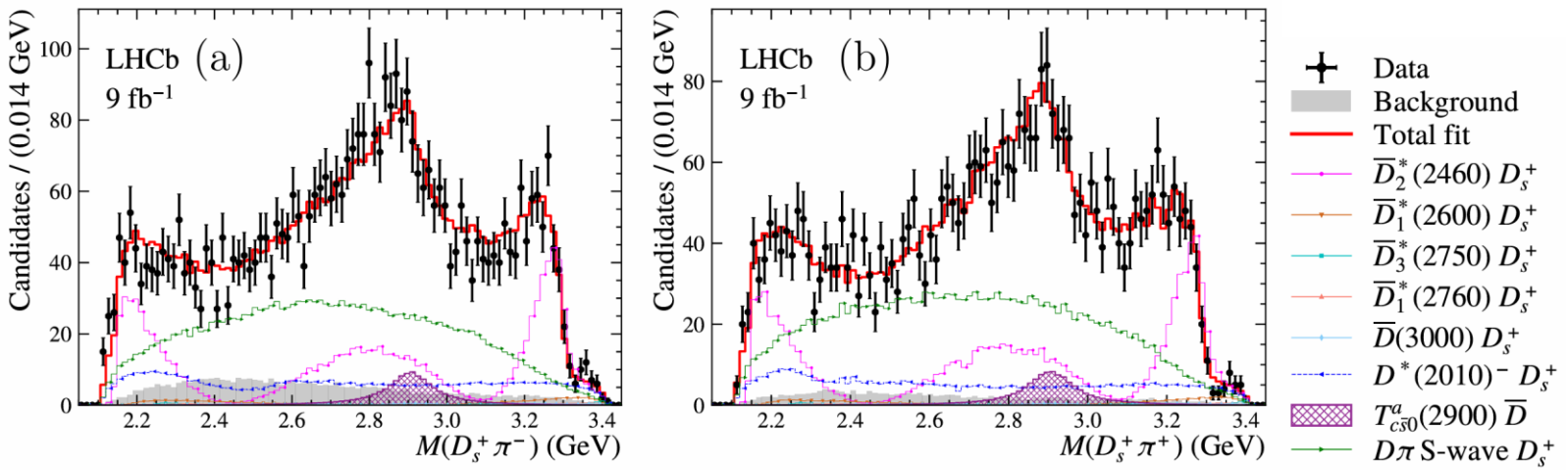


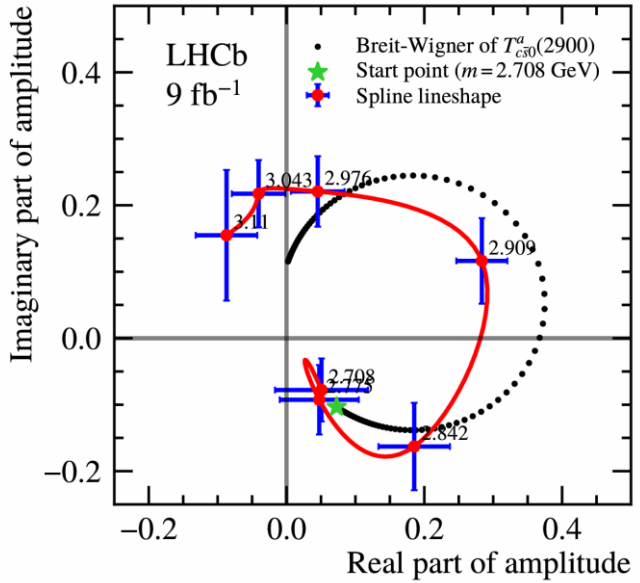
# Example: $B^0 \to \bar{D}^0 D_s^+ \pi^-$, $B^+ \to D^- D_s^+ \pi^+$

PRL.131.041902

PRD.108.012017

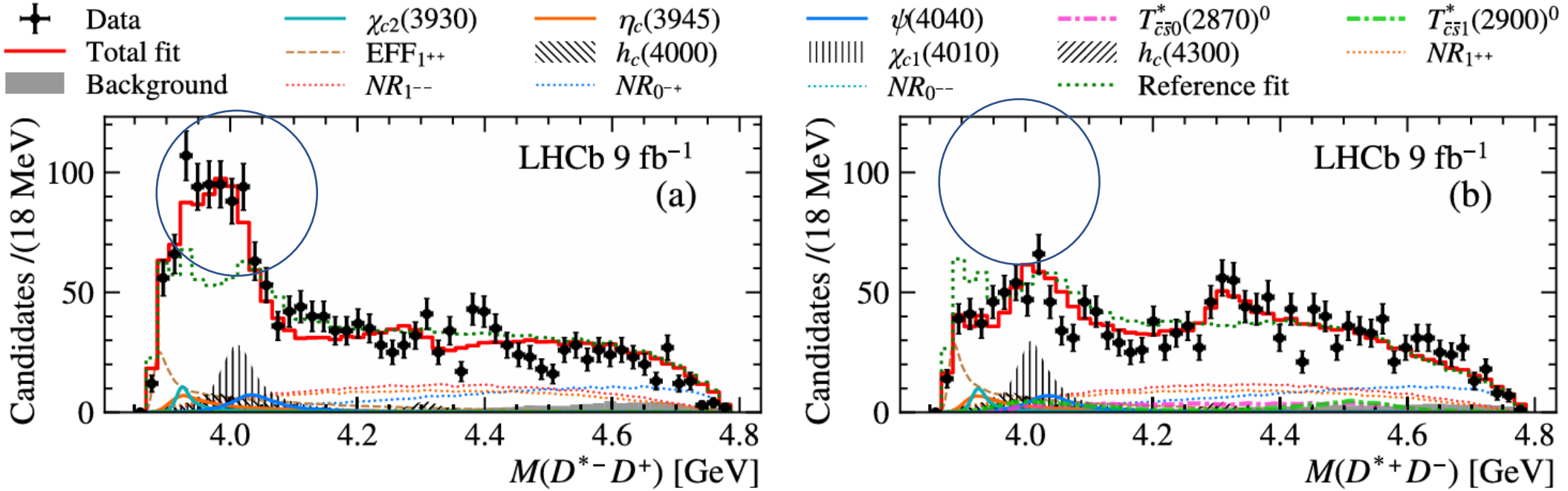
- Constrains with isospin
  - Normal states: $B \to D^* D_s, D^* \to D\pi$
  - Exotic states: $B \to DX, X \to D_s \pi$
- New doubly charged tetraquark candidate and its natural partner
  - $T_{c\bar{s}}^a(2900)^{++,0} \to D_s^+ \pi^\pm$
  - Validation with quasi-model-independent fit
    - Fit with spline model, Argand plot consistent with Breit Wigner circle



# Example: $B^+ \to \mathrm{D}^{*\pm}\mathrm{D}^{\mp}K^+$ PRL.133.131902

- Combine fit with two C-parity related decay channels
  - $B^+ \to \mathrm{D}^{*+}\mathrm{D}^{-}K^+$ and $B^+ \to \mathrm{D}^{*-}\mathrm{D}^{+}K^+$
  - Constraints: $A(B^+ \to R \to (\mathrm{D}^{*+}\mathrm{D}^{-})K^+) = C_R A(B^+ \to R \to (\mathrm{D}^{*-}\mathrm{D}^{+})K^+)$
  - Using the interference to describe the difference in $M(D^*D)$
  - Implemented by custom models to deal with extra information (charges)
- Obverse new states decay to $\mathrm{D}^{*\pm}\mathrm{D}^{\mp}$, and determinate its C-parity.

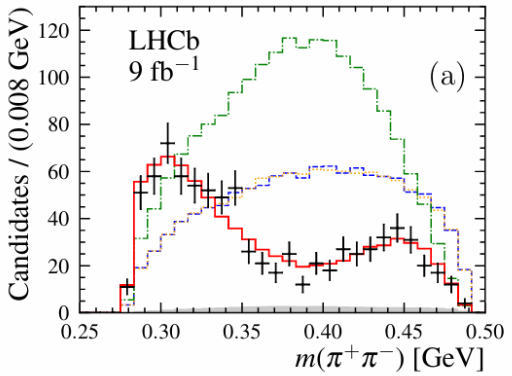


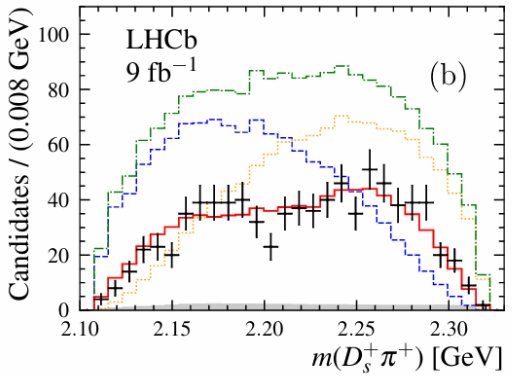
# Example: $B \to D^{(*)} D_{s1}(2460)^+ (\to D_s^+ \pi^+ \pi^-)$

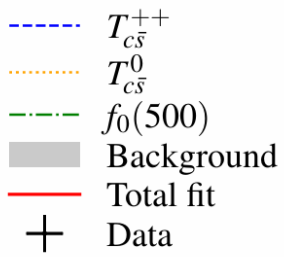
[arxiv:2411.03399]

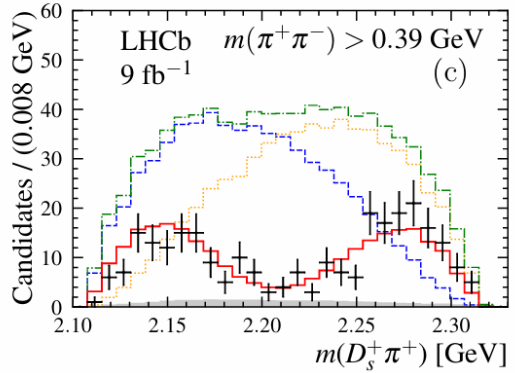
- Simultaneous fit of 3 channels
  - $B^+ \to \bar{D}^0 D_{s1}(2460)^+, B^0 \to D^- D_{s1}(2460)^+, B^0 \to D^{*-} D_{s1}(2460)^+$
  - Share the same model in $D_{s1}(2460)^+ \to D_s^+ \pi^+ \pi^-$
- Test of difference models
  - Some special models
    - Chiral dynamics

      Commun.Theor.Phys. 75 (2023) 5, 055203
      - predict a double-bump lineshape
      - Implement with interpolation
        - $(\lambda_{D_{s1}}, m, \cos\theta)$
    - K-matrix $T_{c\bar{s}}$ (right plot)
      - $K = \begin{pmatrix} \gamma & \beta \\ \beta & \gamma_2 \end{pmatrix}$
      - $f(m) = \frac{\beta^2 \rho_{DK} + i\gamma_2 (i\gamma \rho_{DK} - 1)}{\beta^2 \rho_{DK} \rho_{D_s\pi} + (i\gamma \rho_{DK} - 1)(i\gamma_2 (\rho_{D_s\pi} - 1)}$



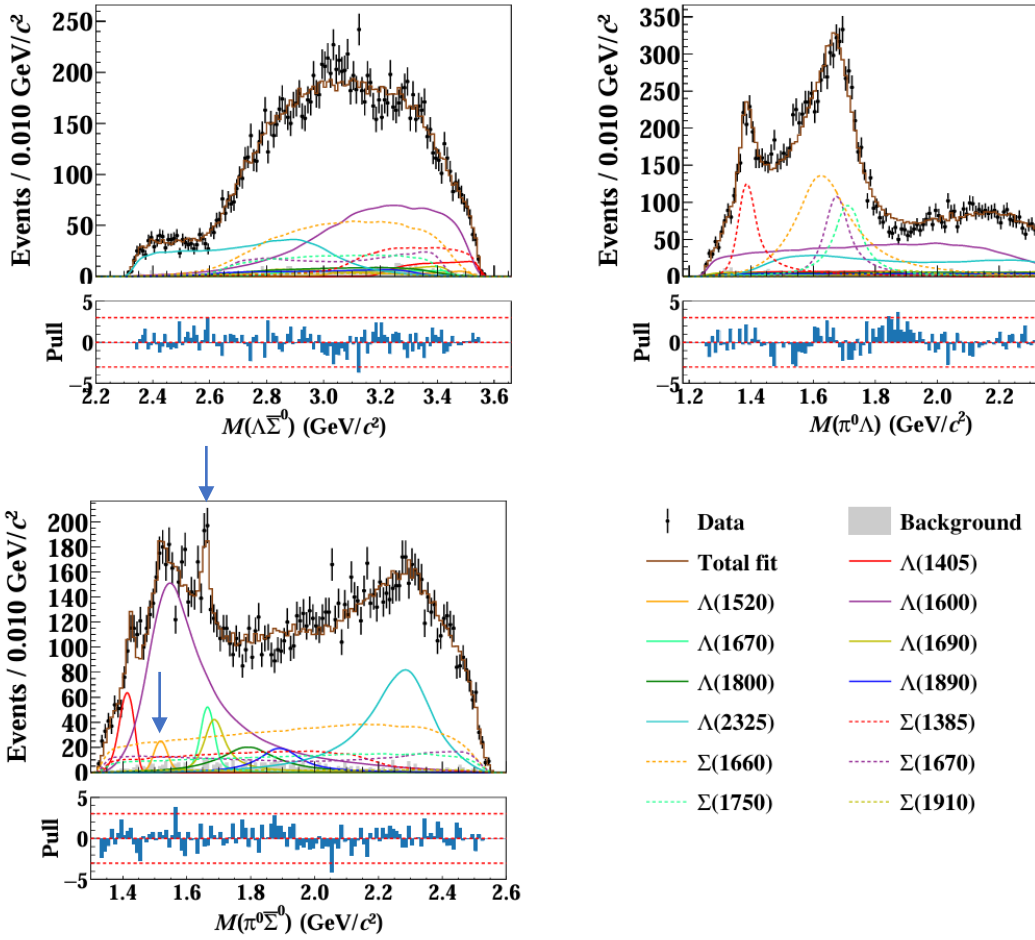



# Example: $\psi(3686) \to \Lambda\bar{\Sigma}^0\pi^0$ JHEP 02 (2025) 212

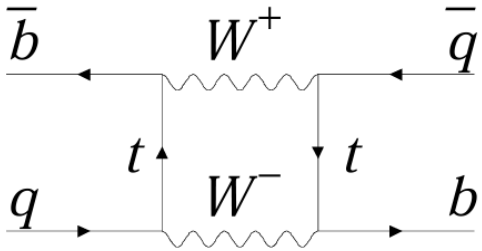
- Double baryon final states
- Consider mass resolution in $M(\pi^0\bar{\Sigma}^0)$
  - Mass resolution: 4.8 MeV
  - Narrow Peak:
    - $\Lambda(1670) \sim 30$ MeV
    - $\Lambda(1520) \sim 20$ MeV
  - $P(m, \cos\theta) \propto |A(m, \cos\theta)|^2 \otimes R(m)$
  - Using numerical integration
    - $m'_k = m + \frac{2k-1}{2N}(\delta_{right} - \delta_{left}) + \delta_{left}$
    - $P(m, \cos\theta) \propto \sum_k |A(m'_k, \cos\theta)|^2 \mathcal{G}(m - m'_k)$
  - Plot fit projection
    - Weight with MC truth value $|A(m, \cos\theta)|^2$
    - Plot the distribution of reconstruction masses with weights.

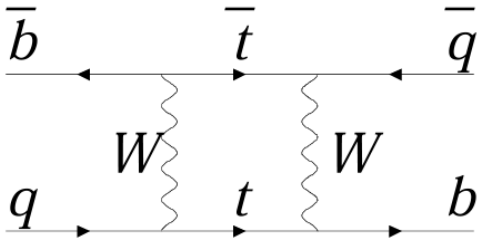



More details of resolution in https://github.com/jiangyi15/tf-pwa/tree/dev/checks/resolution

# Current progress of TF-PWA

- Time dependent Amplitude (WIP)
  - Time dependence #168 #179
    - $|\mathrm{B(t)}\rangle = g_+(t)|B\rangle + \frac{q}{p} g_-(t)|\bar{B}\rangle$
    - $\left|\bar{B}\mathrm{(t)}\right\rangle = \frac{p}{q} g_-(t)|B\rangle + g_+(t)|\bar{B}\rangle$




  - Flavour tagging #183
    - $P(tag, p^\mu, t) = P(tag|B)P(B, p^\mu, t) + P(tag|\bar{B})P(\bar{B}, p^\mu, t)$
  - Production asymmetry #181
    - $P(B, p^\mu, t) = (1 - A_p)|A(t)|^2,\ P(\bar{B}, p^\mu, t) = (1 + A_p)|\bar{A}(t)|^2,$
  - Time resolution, #184
    - $P(B, p^\mu, t) \to \int P(B, p^\mu, t')R(t - t')dt'$
  - Validations (with LHCb)
- Other related updates on framework

# Related project of TF-PWA

- Examples collections
  - Code: https://github.com/jiangyi15/tf-pwa-example
  - A collections PWA models
    - Directly using Using TF-PWA
    - Reproduce trough TF-PWA
  - Repeatability of analysis results
  - Provide examples for TF-PWA
- Custom OP for TensorFlow
  - Code: https://github.com/jiangyi15/tf-pwa-op
  - To provide faster fuse ops in TensorFlow for PWA
- Interface of EvtGen
  - Code: https://github.com/jiangyi15/tf-pwa-evtgen
  - Provide different methods (with and without TensorFlow)
  - Validated in BESEvtGen

# Summary

- TF-PWA
  - Convenient configuration, general proposed
  - Use powerful AD in fitting and error propagation.
  - Provide options to achieve high performance
- Some special examples of TF-PWA
- Current development of TF-PWA.

Thank you for your attentions!

# Backup

-

# 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
  - $\int |A|^2(x) \int \epsilon_y(y) R_y(x \to y) dy dx = \int |A|^2(x) \epsilon_x(x) dx$.

# 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

# 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:

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

      Allow fit parameters related
    - factor only: (only for MC integration)
      - $\sum |A|^2 \rightarrow G_i G_j^* (\sum C_{ij})$

        calculate only once
      - Required all shape parameters fixed
  - Special in simultaneous fit
    - Mixed likelihood, avoid small size data

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

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

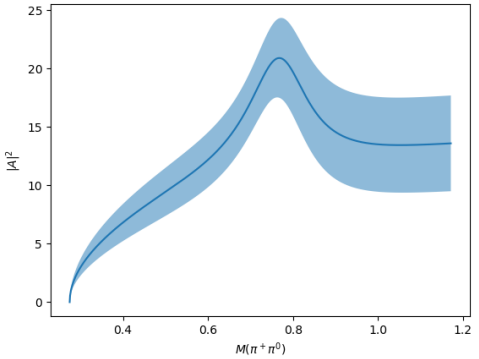
# Amplitude as a function

- 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 $\theta$

```
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



# 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)
```

Define a custom model is very 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)
- $\vartheta$: 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.

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

# 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

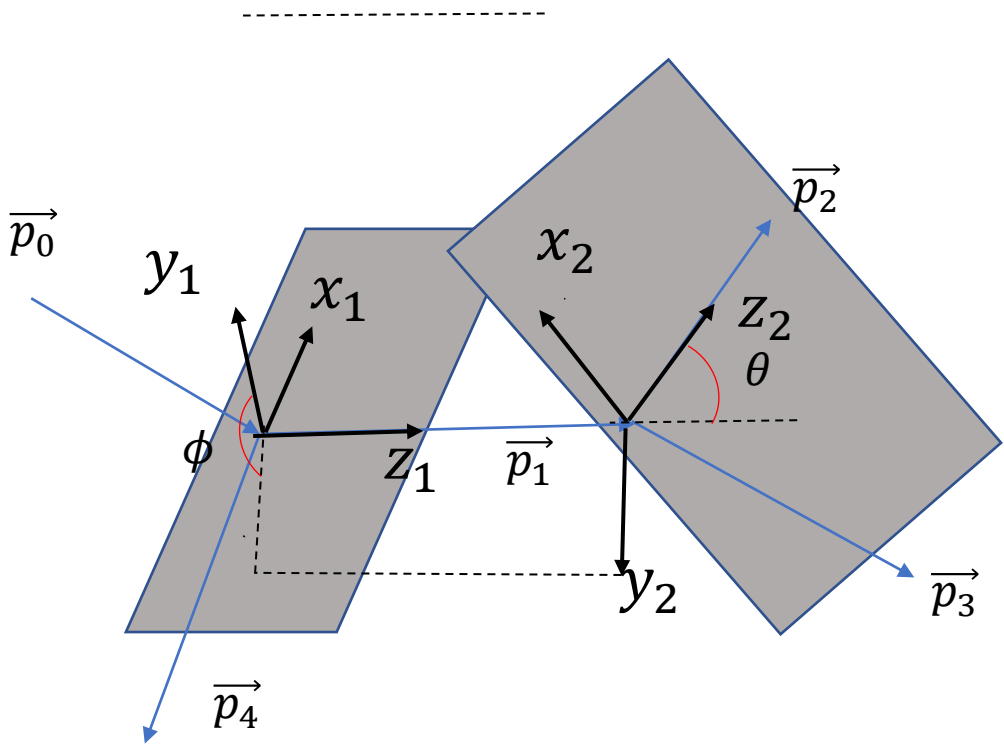
$$|p_1\rangle_2 = L_r|p_1\rangle_1 = R_z(\gamma)R_y(\beta)R_z(\alpha)|p_1\rangle_1$$

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;
```

Output: 1.44329e-15

$x + \sin x$

$grad(Add, Var)$
$grad(Var, x)$
$+grad(Add, \sin(x))$
$grad(\sin(x), Var)$
$grad(\mathrm{Var}, \mathrm{x})$

$\approx 1 + \cos \pi$

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

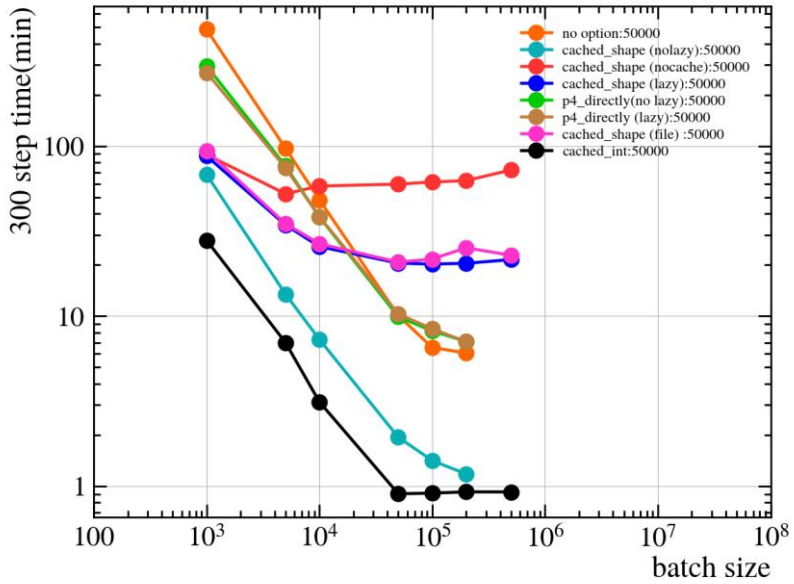
- Option in config.yml:data: model
- Default
  - $-\ln L = -\sum_{i\in data} w_i \ln|A|^2(x_i) + \left(\sum_{i\in data} w_i\right)\ln I_{sig}$
  - $I_{sig} = \frac{\sum_{j\in MC}\omega_j|A|^2(x_j)}{\sum_{j\in MC}\omega_j}$
  - bg will be merged into data with -weights
- cfit:
  - Additional information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

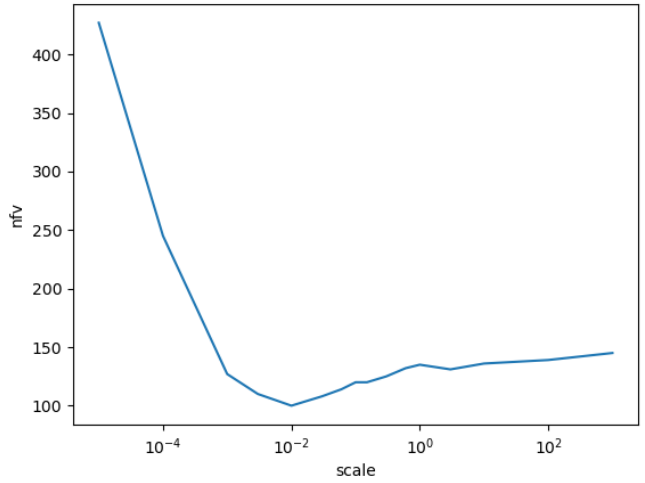
```
data:
  dat_order: [B, C, D]
  data: [data.dat]
  bg: [bg.dat]
  bg_weight: 0.1
  phsp: [phsp.dat]
```

```
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: $\frac{\partial^2 \ln L(\vartheta)}{\partial \vartheta_i \partial \vartheta_j} = \frac{\partial^2 \ln L(\vartheta')}{\partial \vartheta_k' \partial \vartheta_l'}\frac{\partial \vartheta_k'}{\partial \vartheta_i}\frac{\partial \vartheta_l'}{\partial \vartheta_j} + \frac{\partial^2 \ln L(\vartheta')}{\partial I}\frac{\partial^2 I}{\partial \vartheta_i \partial \vartheta_j}$

# Different location for preprocess data

- No cached

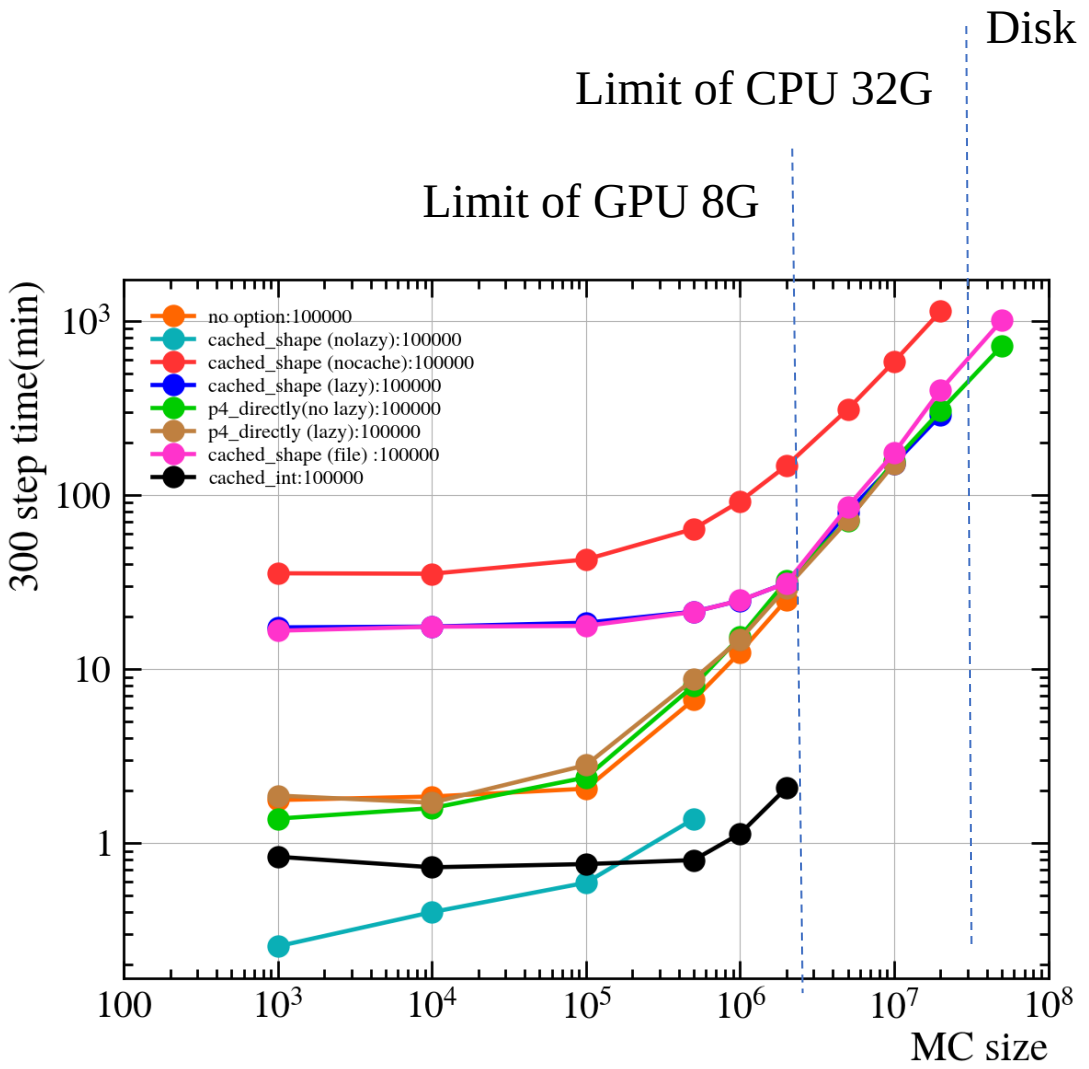
```
data:
  …
  lazy_call: True
```

- GPU vRAM
  - No addition option
- CPU RAM

```
data:
  …
  lazy_call: True
  cached_lazy_file: “”
```

- Disk

```
data:
  …
  lazy_call: True
  cached_lazy_file: dir_of_files/
```

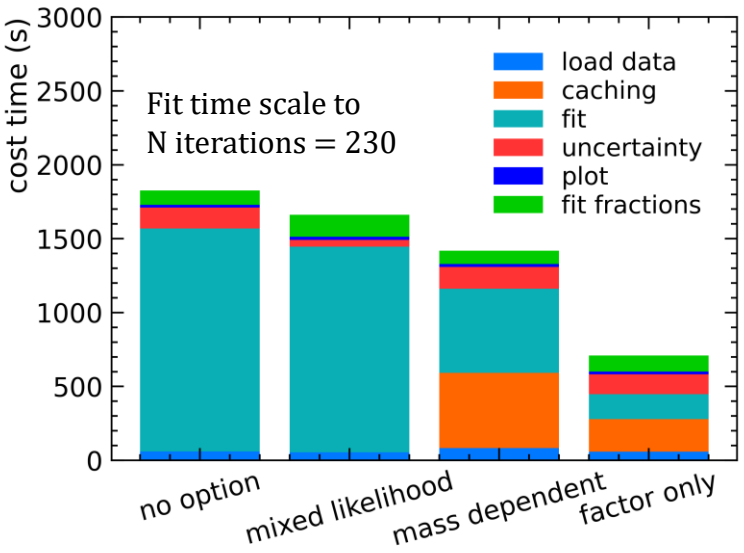


# Real analysis performance

Environment:
NVIDIA RTX 3080
TensorFlow 2.2
CUDA 10.1

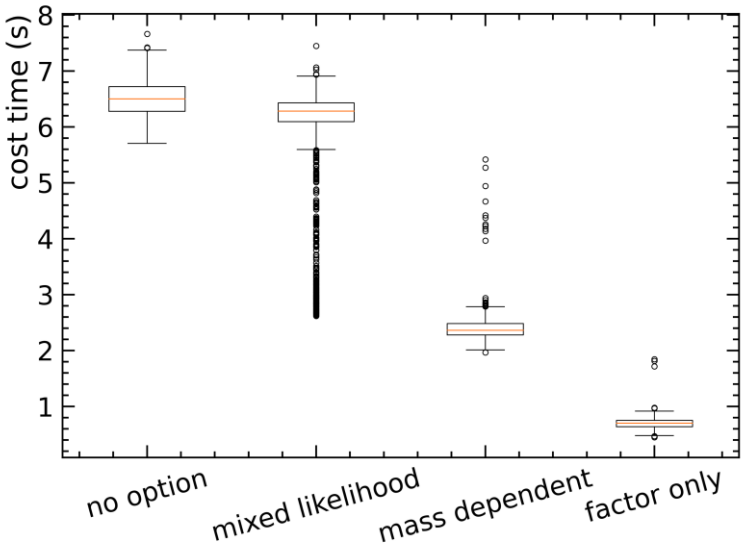
- Optimized method in Factor System page
  - Caching method
    - Large time for caching
    - required more memory
    - limited to special cases
  - All the process is automatic (from config.yml to all basic results)

Total fit time



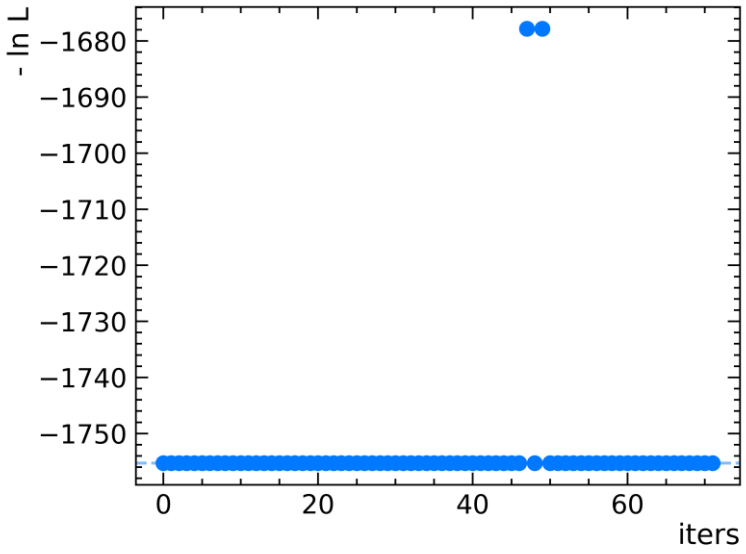

Only half of hours
for ALL the process

Time in each iterations




Caching provide 8 times speed up
for fit parts

Fit stability test




Fit stability looks well
in random initial parameters