
TFPWA

Yi Jiang

Feb 10, 2024

CONTENTS:

1	Install	3
1.1	1. vitrual environment	3
1.2	2. tensorflow2	4
1.3	3. Other dependences	5
1.4	4. (option) Other dependences.	6
1.5	5. Other install method.	6
1.6	6. For developer	6
2	Amplitude	7
2.1	Helicity Formula	7
3	Custom Model	9
3.1	Simple Resonance (experimental)	10
4	Decay Topology	11
4.1	Topology identity: The combination of final particles	11
4.2	From particles to enumerate all possible decay chain topology:	11
5	Resonances Parameters	13
5.1	Common Parameters	13
5.2	Model Parameters	14
5.3	Other Parameters	14
6	Phase Space	15
6.1	Phase Space Generator	16
7	Resolution in Partial Wave Analysis	19
8	Some examples	21
8.1	Examples for particle model	21
8.2	Examples for Plotter class	23
8.3	Particle and amplitude	27
8.4	Examples for error propagation	29
8.5	Examples for config.yml file	32
9	Setup for Developer Enveriment	39
10	API	41
10.1	tf_pwa	41
11	Available Resonances Model	167

12 Available Decay Model	189
12.1 2-body decays	189
13 Tensorflow and Cudatoolkit Version	191
14 FAQ	193
14.1 1. Precision Loss	193
14.2 2. NaN value in fit	193
14.3 3. NaN value when getting params error.	194
14.4 4. Singular Matrix when getting params error	194
14.5 5. Out of memory (OOM)	194
14.6 6. Bad config.yml	195
15 For starters	197
16 Indices and tables	199
Python Module Index	201
Index	203

TFPWA is a generic software package intended for Partial Wave Analysis (PWA). It is developed using TensorFlow2 and the calculation is accelerated by GPU. Users may modify the configuration file (in YAML format) and write simple scripts to complete the whole analysis. A detailed configuration file sample (with all usable parameters) can be found **here**.

INSTALL

To use TFPWA, we need some dependent packages. There are two main ways, `conda` and `virtualenv` you can choose *one* of them. Or you can choose other method in [5. Other install method](#).

1.1 1. vitrual environment

To avoid conflict of dependence, we recommed to use vitrual environment. there are two main vitrual environment for python packages, `conda` and `virtualenv`. You can choose one of them. Since `conda` include `cuda` toolkit for `gpu`, we recommed it for user.

1.1.1 1.1 conda

- 1.1.1 Get miniconda for python3 from [miniconda3](#) and install it.
- 1.1.2 Create a virtual environment by

```
conda create -n tfpwa
```

, the `-n <name>` option will create a environment named by `<name>`. You can also use `-p <path>` option to create environment in the `<path>` directory.

- 1.1.3 You can activate the environment by

```
conda activate tfpwa
```

and then you can install packages in the `conda` environment

- 1.1.4 You can exit the environment by

```
conda deactivate
```

1.1.2 1.2 virtualenv

- 1.2.1 You should have a python3 first.
- 1.2.2 Install virtualenv

```
python3 -m pip install --user virtualenv
```

- 1.2.3 Create a virtual environment

```
python3 -m virtualenv ./tfpwa
```

, it will store in the path `tfpwa`

- 1.2.4 You can activate the environment by

```
source ./tfpwa/bin/activate
```

- 1.2.5 You can exit the environment by

```
deactivate
```

1.2 2. tensorflow2

The most important package is [tensorflow2](#). We recommend to install tensorflow first. You can follow the install instructions in [tensorflow website](#) (or [tensorflow.org](#)).

1.2.1 2.1 conda

Here we provide the simple way to install tensorflow2 gpu version in conda environment

```
conda install tensorflow-gpu=2.4
```

it will also install cudatoolkit.

1.2.2 2.2 virtualenv

When using virtualenv, there is also a simple way to install tensorflow2

```
python -m pip install tensorflow
```

, but you should check your CUDA installation for GPU.

Note: You can use `-i https://pypi.tuna.tsinghua.edu.cn/simple` option to use pypi mirror site.

1.3 3. Other dependences

Other dependences of TFPWA is simple.

1.3.1 3.1 Get TFPWA package

Get the packages using

```
git clone https://github.com/jiangyi15/tf-pwa
```

1.3.2 3.2 conda

3.2.1 other dependences

In conda environment, go into the directory of `tf-pwa`, you can install the rest dependences by

```
conda install --file requirements-min.txt
```

Note: we recommend Ampere card users to install with `tensorflow_2_6_requirements.txt` (see this [technical FAQ](#)).

```
conda install --file tensorflow_2_6_requirements.txt -c conda-forge
```

3.2.2 TFPWA

install TFPWA

```
python -m pip install -e ./ --no-deps
```

Use `--no-deps` to make sure that no PyPI package will be installed. Using `-e`, so it can be updated by `git pull` directly.

1.3.3 3.3 virtualenv

In virtualenv, You can install dependences and TFPWA together.

```
python3 -m pip install -e ./
```

Using `-e`, so it can be updated by `git pull` directly.

1.4 4. (option) Other dependences.

There are some option packages, such as `uproot` for reading root file.

1.4.1 4.1 conda

It can be installed as

```
conda install uproot -c conda-forge
```

1.4.2 4.2 virtualenv

It can be installed as

```
python -m pip install uproot
```

1.5 5. Other install method.

We also provided other install method.

1.5.1 5.1 conda channel (experimental)

A pre-built conda package (Linux only) is also provided, just run following command to install it.

```
conda config --add channels jiangyi15  
conda install tf-pwa
```

1.5.2 5.2 pip

When using `pip`, you will need to install CUDA to use GPU. Just run the following command :

```
python3 -m pip install -e .
```

1.6 6. For developer

To contribute to the project, please also install additional developer tools with:

```
python3 -m pip install -e .[dev]
```

AMPLITUDE

2.1 Helicity Formula

Each Decay has Amplitude like

$$A_{\lambda_A, \lambda_B, \lambda_C}^{A \rightarrow B+C} = H_{\lambda_B, \lambda_C}^{A \rightarrow B+C} D_{\lambda_A, \lambda_B - \lambda_C}^{J_A^*}(\phi, \theta, 0)$$

For a chain decay, amplitude can be combined as

$$A_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{A \rightarrow R+B, R \rightarrow C+D} = \sum_{\lambda_R} A_{\lambda_A, \lambda_R, \lambda_B}^{A \rightarrow R+B} R(m_R) A_{\lambda_R, \lambda_C, \lambda_D}^{R \rightarrow C+D}$$

with angle aligned

$$\hat{A}_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{A \rightarrow R+B, R \rightarrow C+D} = \sum_{\lambda'_B, \lambda'_C, \lambda'_D} A_{\lambda_A, \lambda'_B, \lambda'_C, \lambda'_D}^{A \rightarrow R+B, R \rightarrow C+D} D_{\lambda'_B, \lambda_B}^{J_B^*}(\alpha_B, \beta_B, \gamma_B) D_{\lambda'_C, \lambda_C}^{J_C^*}(\alpha_C, \beta_C, \gamma_C) D_{\lambda'_D, \lambda_D}^{J_D^*}(\alpha_D, \beta_D, \gamma_D)$$

the sum of resonances

$$A_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{total} = \sum_{R_1} \hat{A}_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{A \rightarrow R_1+B, R_1 \rightarrow C+D} + \sum_{R_2} \hat{A}_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{A \rightarrow R_2+C, R_2 \rightarrow B+D} + \sum_{R_3} \hat{A}_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{A \rightarrow R_3+D, R_3 \rightarrow B+C}$$

then the differential cross-section

$$\frac{d\sigma}{d\Phi} = \frac{1}{N} \sum_{\lambda_A} \sum_{\lambda_B, \lambda_C, \lambda_D} |A_{\lambda_A, \lambda_B, \lambda_C, \lambda_D}^{total}|^2$$

2.1.1 Amplitude Combination Rules

For a decay process $A \rightarrow R B$, $R \rightarrow C D$, we can get different part of amplitude:

1. Particle:

1. Initial state: 1
2. Final state: $D(\alpha, \beta, \gamma)$
3. Propagator: $R(m)$

2. Decay:

Two body decay ($A \rightarrow R B$): $H_{\lambda_R, \lambda_B} D_{\lambda_A, \lambda_R - \lambda_B}(\varphi, \theta, 0)$

Now we can use combination rules to build amplitude for the whole process.

Probability Density:

$$P = |\tilde{A}|^2 \text{ (modular square)}$$

Decay Group:

$$\tilde{A} = a_1 A_{R_1} + a_2 A_{R_2} + \cdots \text{ (addition)}$$

Decay Chain:

$$A_R = A_1 \times R \times A_2 \cdots \text{ (multiplication)}$$

$$\text{Decay: } A_i = HD(\varphi, \theta, 0)$$

$$\text{Particle: } R(m)$$

The indices part is quantum number, and it can be summed automatically.

2.1.2 Default Amplitude Model

The default model for Decay is helicity amplitude

$$A_{\lambda_A, \lambda_B, \lambda_C}^{A \rightarrow BC} = H_{\lambda_B, \lambda_C}^{A \rightarrow BC} D_{\lambda_A, \lambda_B - \lambda_C}^{J_A^*}(\phi, \theta, 0).$$

The LS coupling formula is used

$$H_{\lambda_B, \lambda_C}^{A \rightarrow B+C} = \sum_{ls} g_{ls} \sqrt{\frac{2l+1}{2J_A+1}} \langle l0; s\delta | J_A \delta \rangle \langle J_B \lambda_B; J_C - \lambda_C | s\delta \rangle q^l B_l'(q, q_0, d)$$

g_{ls} are the fit parameters, the first one is fixed. q and q_0 is the momentum in rest frame for invariant mass and resonance mass.

$B_l'(q, q_0, d)$ (*Bprime*) is Blatt-Weisskopf barrier factors. d is 3.0GeV^{-1} by default.

Resonances model use Relativistic Breit-Wigner function

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

with running width

$$\Gamma(m) = \Gamma_0 \left(\frac{q}{q_0} \right)^{2L+1} \frac{m_0}{m} B_L'^2(q, q_0, d).$$

By using the combination rules, the amplitude is built automatically.

CUSTOM MODEL

TF-PWA support custom model of *Particle*, just implement the *Particle.get_amp* method for a class inherited from *Particle* as:

```
from tf_pwa.amp import register_particle, Particle

@register_particle("MyModel")
class MyParticle(Particle):
    def get_amp(self, *args, **kwargs):
        print(args, kwargs)
        return 1.0
```

Then it can be used in config.yml (or Resonances.yml) as model: MyModel. We can get the data used for amplitude, and add some calculations such as Breit-Wigner.

```
from tf_pwa.amp import register_particle, Particle
import tensorflow as tf

@register_particle("BW")
class SimpleBW(Particle):
    def get_amp(self, *args, **kwargs):
        """
        Breit-Wigner formula
        """
        m = args[0]["m"]
        m0 = self.get_mass()
        g0 = self.get_width()
        delta_m = m0*m0 - m * m
        one = tf.ones_like(m)
        ret = 1/tf.complex(delta_m, m0 * g0 * one)
        return ret
```

Note, we used one to make sure the shape to be same.

We can also add parameters in the *Model* *init_params* using *self.add_var(...)*.

```
@register_particle("Line")
class LineModel(Particle):
    def init_params(self):
        super(LineModel, self).init_params()
        self.a = self.add_var("a")
    def get_amp(self, data, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
""" model as m + a """
m = data["m"]
zeros = tf.zeros_like(m)
return tf.complex( m + self.a(), zeros)
```

Then a parameters {particle_name}_a will appear in the parameters, we use `self.a()` to get the value in `get_amp`. Note, the type of return value should be `tf.complex`. All builtin model is located in `tf_pwa/amp.py`.

3.1 Simple Resonance (experimental)

There is a simple method to define Resonance model, like

```
from tf_pwa.amp import simple_resonance, FloatParams

@simple_resonance("Line_2", params=["a"])
def r_line(m, a: FloatParams = 1.0):
    return m + a
```

Those code will build a class similar as Line model define before. By using `inspect` module, we can get the `FullArgSpec` of a function. For a keyword arguments with type annotation as `FloatParams`, it will be treated as a fit parameters.

Note: the first arguments have to be the invariant mass `m` of the resonance.

DECAY TOPOLOGY

A decay chain is a simple tree, from top particle to final particles. So the decay chain can be describing as Node (*Decay*) and Line (*Particle*)

4.1 Topology identity: The combination of final particles

For example, the combination of decay chain $A \rightarrow RC, R \rightarrow BD$ and $A \rightarrow ZB, R \rightarrow CD$ is

```
{A: [B, C, D], R: [B, D], B: [B], C: [C], D: [D]}
```

and

```
{A: [B, C, D], Z: [C, D], B: [B], C: [C], D: [D]}
```

The item R and Z is not same, so there are two different topology.

```
{{A: [B, C, D], B: [B], C: [C], D: [D]}}
```

is the direct $A \rightarrow BCD$ decay.

4.2 From particles to enumerate all possible decay chain topology:

From a basic line, inserting lines to create a full graph.

from a line: $A \rightarrow B$,

insert a line ($\text{node0} \rightarrow C$) and a node (node0):

```
1. A -> node0, node0 -> B, node0 -> C
```

insert a line :

```
1. A -> node0, node0 -> B, node0 -> node1, node1 -> C, node1 -> D
2. A -> node1, node1 -> node0, node0 -> B, node0 -> C, node1 -> D
3. A -> node0, node0 -> node1, node1 -> B, node0 -> C, node1 -> D
```

there are the three possible decay chains of $A \rightarrow B,C,D$

1. $A \rightarrow R+B, R \rightarrow C+D$
2. $A \rightarrow R+D, R \rightarrow B+C$
3. $A \rightarrow R+C, R \rightarrow B+D$

the process is unique for different final particles

Each inserting process delete a line and add three new line, So for decay process has n final particles, there are $(2n-3)!!$ possible decay topology.

RESONANCES PARAMETERS

This section is about how do the `Resonances.yml` work.

From `Resonances.yml` to the real model, there will be following steps.

1. loaded by `config.yml`, it is will be combined the defination in `config.yml` particle parts.

For examples, `config.yml` have

```
particle:
  $include: Resonances.yml
  Psi4040:
    float: mg
```

then `Resonances.yml` item

```
Psi4040:
  J: 1
  float: m
```

will become `{"Psi4040": {"J": 1, "float": "mg"}}`

2. replace some alias, (`m0` -> `mass`, `g0` -> `width`, ...)
3. If it is used in decay chain, then create `Particle` object.

The particle class is `cls = get_particle_model(item["model"])`, and the object is `cls(**item)`.

All parameters will be stored in `config.particle_property[name]`.

All aviable parameters can be devied into the flowowing 3 sets.

5.1 Common Parameters

Parameters defined in *BaseParticle* are common parameters including spin, parity, mass and width.

name	default value	comment
J	0	spin, int or half-integral
P	-1	P-parity, +1 or -1
C	None	C-Parity, +1 or -1
mass	None	mass, float, it is always required because of the calculation of q_0
width	None	width, float
spins	None	possible spin projections, <code>[-J, ..., J]</code> , list

5.2 Model Parameters

Parameters defined in the real model. *Available Resonances Model*

There are many parameters defined by the user, the those parameters will be pass to model class, such as the paramthens for `__init__(self, **kwargs)` method.

For examples, the default model (BWR, *BaseParticle*) have following parameters:

name	default value	comment
<code>running_width</code>	True	if using running width, bool
<code>bw_l</code>	None, auto deteminated	running width angular momentum, int

5.3 Other Parameters

There are also some other parameters which is used to control the program running.

For examples, simple constrains, the following parameters are using by `ConfigLoader` as constrains.

name	default value	comment
<code>mass_min, mass_max</code>	None	mass range
<code>width_min, width_max</code>	None	width range
<code>float</code>	[]	float paramsters list

Another examples are parameters to build decay chain for particle R.

name	default value	comment
<code>decay_params</code>	{}	parameters pass to Decay which R decay
<code>production_params</code>	{}	parameters pass to Decay which produce R
<code>model</code>	default	Particle model for R

There are also other common used parameters.

name	default value	comment
<code>display</code>	<code>None</code>	control plot legend with latex string, string

PHASE SPACE

N body decay phase space can be defined as

$$d\Phi(P; p_1, \dots, p_n) = (2\pi)^4 \delta^4(P - \sum p_i) \prod \theta(p_i^0) 2\pi \delta(p_i^2 - m_i^2) \frac{d^4 p_i}{(2\pi)^4}$$

or integrate p^0 as

$$d\Phi(P; p_1, \dots, p_n) = (2\pi)^4 \delta^4(P - \sum p_i) \prod \frac{1}{2E_i} \frac{d^3 \vec{p}_i}{(2\pi)^3}$$

by using the property of δ -function,

$$\delta(f(x)) = \sum_i \frac{\delta(x - x_i)}{|f'(x_i)|}$$

where x_i is the root of $f(x) = 0$.

Phase space has the follow chain rule,

$$\begin{aligned} d\Phi(P; p_1, \dots, p_n) &= (2\pi)^4 \delta^4(P - \sum p_i) \prod \frac{1}{2E_i} \frac{d^3 \vec{p}_i}{(2\pi)^3} \\ &= (2\pi)^4 \delta^4(P - \sum_{i=0}^m p_i - q) \prod_{i=0}^m \frac{1}{2E_i} \frac{d^3 \vec{p}_i}{(2\pi)^3} \prod_{i=m+1}^n \frac{1}{2E_i} \frac{d^3 \vec{p}_i}{(2\pi)^3} \\ &\quad (2\pi)^4 \delta^4(q - \sum_{i=m+1}^n p_i) \frac{d^4 q}{(2\pi)^4} \delta(q^2 - (\sum_{i=m+1}^n p_i)^2) dq^2 \\ &= d\Phi(P; p_1, \dots, p_m, q) \frac{dq^2}{2\pi} d\Phi(q; p_{m+1}, \dots, p_n), \end{aligned}$$

where $q = \sum_{i=m+1}^n p_i$ is the invariant mass of particles $m+1, \dots, n$.

The two body decay is simple in the center mass frame $P = (M, 0, 0, 0)$,

$$\begin{aligned} d\Phi(P; p_1, p_2) &= (2\pi)^4 \delta^4(P - p_1 - p_2) \frac{1}{2E_1} \frac{d^3 \vec{p}_1}{(2\pi)^3} \frac{1}{2E_2} \frac{d^3 \vec{p}_2}{(2\pi)^3} \\ &= 2\pi \delta(M - E_1 - E_2) \frac{1}{2E_1} \frac{1}{2E_2} \frac{d^3 \vec{p}_2}{(2\pi)^3} \\ &= 2\pi \delta(M - \sqrt{|\vec{p}|^2 + m_1^2} - \sqrt{|\vec{p}|^2 + m_2^2}) \frac{1}{2E_1} \frac{|\vec{p}|^2}{2E_2} \frac{d|\vec{p}| d\Omega}{(2\pi)^3} \\ &= \frac{|\vec{p}|}{16\pi^2 M} d\Omega \end{aligned}$$

where $d\Omega = d(\cos \theta)d\varphi$ and

$$E_1 = \frac{M^2 + m_1^2 - m_2^2}{2M}, E_1 = \frac{M^2 - m_1^2 + m_2^2}{2M}$$

$$|\vec{p}| = \frac{\sqrt{(M^2 - (m_1 + m_2)^2)(M^2 - (m_1 - m_2)^2)}}{2M}$$

The three body decay in the center mass frame $P = (M, 0, 0, 0)$, $q^* = (m_{23}, 0, 0, 0)$,

$$\begin{aligned} d\Phi(P; p_1, p_2, p_3) &= d\Phi(P; p_1, q) d\Phi(q^*; p_2^*, p_3^*) \frac{dq^2}{2\pi} \\ &= \frac{|\vec{p}_1| |\vec{p}_2^*|}{(2\pi)^5 16 M m_{23}} dm_{23}^2 d\Omega_1 d\Omega_2^* \\ &= \frac{|\vec{p}_1| |\vec{p}_2^*|}{(2\pi)^5 8 M} dm_{23} d\Omega_1 d\Omega_2^* \end{aligned}$$

The n body decay in the center mass frame $P = (M, 0, 0, 0)$,

$$\begin{aligned} d\Phi(P; p_1, \dots, p_n) &= d\Phi(P; p_1, q_1) \prod_{i=1}^{n-2} \frac{dq_i^2}{2\pi} d\Phi(q_i, p_{i+1}, p_{i+2}) \\ &= \frac{1}{2^{2n-2} (2\pi)^{3n-4}} \frac{|\vec{p}_1|}{M} d\Omega_1 \prod_{i=1}^{n-2} \frac{|\vec{p}_{i+1}^*|}{M_i} dM_i^2 d\Omega_{i+1}^* \\ &= \frac{1}{2^n (2\pi)^{3n-4}} \frac{|\vec{p}_1|}{M} d\Omega_1 \prod_{i=1}^{n-2} |\vec{p}_{i+1}^*| dM_i d\Omega_{i+1}^* \end{aligned}$$

where

$$M_i^2 = (\sum_{j>i} p_j)^2, |\vec{p}_i^*| = \frac{\sqrt{(M_i^2 - (M_{i+1} + m_{i+1})^2)(M_i^2 - (M_{i+1} - m_{i+1})^2)}}{2M_i}$$

with those limit

$$\sum_{j>i} m_j < M_{i+1} + m_{i+1} < M_i < M_{i-1} - m_i < M - \sum_{j\leq i} m_j$$

6.1 Phase Space Generator

For n body phase space

$$d\Phi(P; p_1, \dots, p_n) = \frac{1}{2^n (2\pi)^{3n-4}} \left(\frac{1}{M} \prod_{i=0}^{n-2} |\vec{p}_{i+1}^*| \right) \prod_{i=1}^{n-2} dM_i \prod_{i=0}^{n-2} d\Omega_{i+1}^*,$$

take a weaker condition

$$\sum_{j>i} m_j < M_i < M - \sum_{j\leq i} m_j,$$

has the simple limit at the factor term

$$\begin{aligned} \frac{1}{M} \prod_{i=0}^{n-2} |\vec{p}_{i+1}^*| &= \frac{1}{M} \prod_{i=0}^{n-2} q(M_i, M_{i+1}, m_{i+1}) \\ &< \frac{1}{M} \prod_{i=0}^{n-2} q(\max(M_i), \min(M_{i+1}), m_{i+1}) \end{aligned}$$

- 1. Generate M_i with the factor
- 2. Generate $d\Omega = d\cos\theta d\varphi$
- 3. boost $p^\star = (\sqrt{|\vec{p}^\star|^2 + m^2}, |\vec{p}^\star| \cos\theta \cos\varphi, |\vec{p}^\star| \sin\theta \sin\varphi, |\vec{p}^\star| \cos\theta,)$ to a same frame.

RESOLUTION IN PARTIAL WAVE ANALYSIS

Resolution is the effect of detector. To Consider the resolution properly, We need to take a general look about the detector process. We can divide the process of detector into two parts. The first part is acceptance, with the probability for truth value x as $\epsilon_T(x)$. The second part is resolution, it means the measurement value y will be a random number base on truth value x . It is a conditional probability as $R_T(y|x)$. The conditional probability is normalized as $\int R_T(y|x)dy = 1$. So, the total effect of detector is transition function

$$T(x, y) = R_T(y|x)\epsilon_T(x).$$

When we have a distribution of truth value with probability $p(x)$, then we can get the distribution of measurement value with probability

$$p'(y) = \int p(x)T(x, y)dx.$$

Using the *Bayes Rule*, we can rewrite $T(x, y)$ as

$$T(x, y) = R(x|y)\epsilon_R(y),$$

where

$$\epsilon_R(y) = \int T(x, y)dx, \quad R(x|y) = \frac{T(x, y)}{\epsilon_R(y)}.$$

$R(x|y)$ is the posterior probability, that means the probability of a certain y is from x . $\epsilon_R(y)$ is the projection of y for $T(x, y)$, and is also the normalize factor for $R(x|y)$.

Then, the probability $p'(y)$ can be rewritten as

$$p'(y) = \epsilon_R(y) \int p(x)R(x|y)dx.$$

To consider the resolution, we need to determine $R(x|y)$. Generally, we use simulation to determine $R(x|y)$. When $p(x) = 1$ is a flat distribution, then the joint distribution of x and y has the probability density $T(x, y)$. We can build a model for this distribution. To get $R(x|y)$, we only need to do a normalization for $T(x, y)$.

In PWA, we usually use the MC to do the normalization for signal probability density. We need to calculate the integration of $p'(y)$ as

$$\int p'(y)dy = \int p(x)\epsilon_T(x) \int R_T(y|x)dydx = \int p(x)\epsilon_T(x)dx.$$

The final negative log-likelihood with considering resolution is

$$-\ln L = -\sum \ln \frac{p'(y)}{\int p'(y)dy} = -\sum \ln \frac{\int p(x)R(x|y)dx}{\int p(x)\epsilon_T(x)dx} - \sum \ln \epsilon_R(y).$$

The last part is a constant, we can ignore it in fit. In the numerical form, it can be written as

$$-\ln L = -\sum \ln \frac{1}{M} \sum_{x \sim R(x|y)} p(x) + N \ln \sum_{x \sim \epsilon_T(x)} p(x).$$

For the second part, which we already have MC sample with $x \sim \epsilon_T(x)$, we can use MC sample to do the sum directly. For the first part, we can generate some x (M times) for every y (N events). Using the generated samples (MN events), we can calculate though the summation.

In addition, we can insert some importance information for the summation as

$$\int p(x) R(x|y) dx \approx \frac{1}{\sum w_i} \sum_{x \sim \frac{R(x|y)}{w_i(x)}} w_i p(x).$$

We need to keep the normalization. For example, we can use Gauss-Hermite quadrature.

In a simple situation, we only use mass for the variable for resolution function. We can build the datasets by replacing the mass by random number based on the resolution function, keeping the same for other variables and using some constrains.

Once we get such datasets, we can use the likelihood method to fit the dataset with resolution. There is an example in [checks](#).

SOME EXAMPLES

8.1 Examples for particle model

decay system is model as

DecayGroup
DecayChain
Decay
Particle

```
16 import matplotlib.pyplot as plt
17
18 from tf_pwa.amp import Decay, DecayChain, DecayGroup, Particle
19 from tf_pwa.vis import plot_decay_struct
```

We can easy create some instance of Particle and then combine them as Decay

```
25 a = Particle("A")
26 b = Particle("B")
27 c = Particle("C")
28 d = Particle("D")
29
30 r = Particle("R")
31
32 dec1 = Decay(a, [r, b])
33 dec2 = Decay(r, [c, d])
```

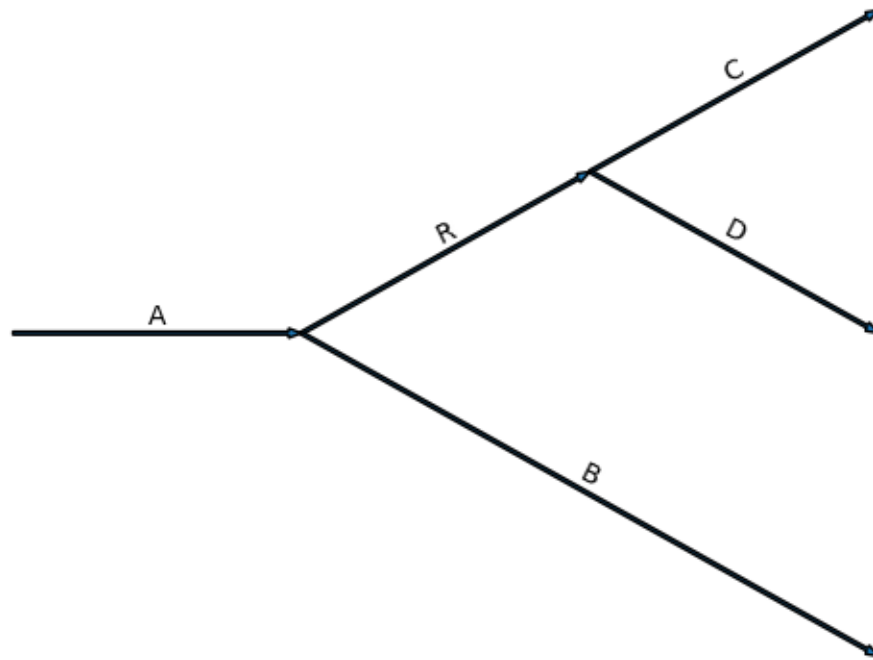
DecayChain is a list of Decays.

```
39 decay_chain = DecayChain([dec1, dec2])
40 decay_chain
```

```
[A->R+B, R->C+D]
```

We can plot it using matplotlib.

```
45 plot_decay_struct(decay_chain)
46 plt.show()
```



DecayGroup is a list of DecayChain with the same initial and final states

```
51 decay_group = DecayGroup([decay_chain])
52 decay_group
```

```
[[A->R+B, R->C+D]]
```

We can build a simple function to infer the charge from final states.

```
58 def charge_infer(dec, charge_map):
59     # use particle equal condition
60     cached_charge = {Particle(k): v for k, v in charge_map.items()}
61     # loop for all decays in decay chain
62     for i, dec_i in dec.depth_first(False):
63         # all out particles has charge
64         assert all(i in cached_charge for i in dec_i.outs)
65         # the charge or core particle is the sum of
66         cached_charge[dec_i.core] = sum(cached_charge[i] for i in dec_i.outs)
67     return cached_charge
68
69
70 charges = {
71     "B": -1,
72     "C": 0,
```

(continues on next page)

(continued from previous page)

```

73     "D": 1,
74 }
75
76 charge_infer(decay_chain, charges)

```

```
{B: -1, C: 0, D: 1, R: 1, A: 0}
```

See more in [cal_chain_boost](#).

Total running time of the script: (0 minutes 0.117 seconds)

8.2 Examples for Plotter class

Plotter is the new api for partial wave plots.

First, we can build a simple config.

```

12 config_str = """
13
14 decay:
15     A:
16         - [R1, B]
17         - [R2, C]
18         - [R3, D]
19     R1: [C, D]
20     R2: [B, D]
21     R3: [B, C]
22
23 particle:
24     $top:
25         A: { mass: 1.86, J: 0, P: -1}
26     $finals:
27         B: { mass: 0.494, J: 0, P: -1}
28         C: { mass: 0.139, J: 0, P: -1}
29         D: { mass: 0.139, J: 0, P: -1}
30     R1: [ R1_a, R1_b ]
31     R1_a: { mass: 0.7, width: 0.05, J: 1, P: -1}
32     R1_b: { mass: 0.5, width: 0.05, J: 0, P: +1}
33     R2: { mass: 0.824, width: 0.05, J: 0, P: +1}
34     R3: { mass: 0.824, width: 0.05, J: 0, P: +1}
35
36
37 plot:
38     mass:
39         R1:
40             display: "m(R1)"
41         R2:
42             display: "m(R2)"
43
44
45 import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```

46 import yaml
47
48 from tf_pwa.config_loader import ConfigLoader
49
50 config = ConfigLoader(yaml.full_load(config_str))

```

We set parameters to a blance value. And we can generate some toy data and calculte the weights

```

56 input_params = {
57     "A->R1_a.BR1_a->C.D_total_0r": 6.0,
58     "A->R1_b.BR1_b->C.D_total_0r": 1.0,
59     "A->R2.CR2->B.D_total_0r": 2.0,
60     "A->R3.DR3->B.C_total_0r": 1.0,
61 }
62 config.set_params(input_params)
63
64 data = config.generate_toy(1000)
65 phsp = config.generate_phsp(10000)

```

```

7.5%[>-----] 1.02/13.57s eff: 90.000000%
74.3%[>-----] 3.05/4.10s eff: 6.214227%
102.6%[>] 4.21/4.10s eff: 4.228234%
100.0%[] 4.21/4.21s eff: 4.229470%

```

plotter can be created directly from config

```

71 plotter = config.get_plotter(datasets={"data": [data], "phsp": [phsp]})

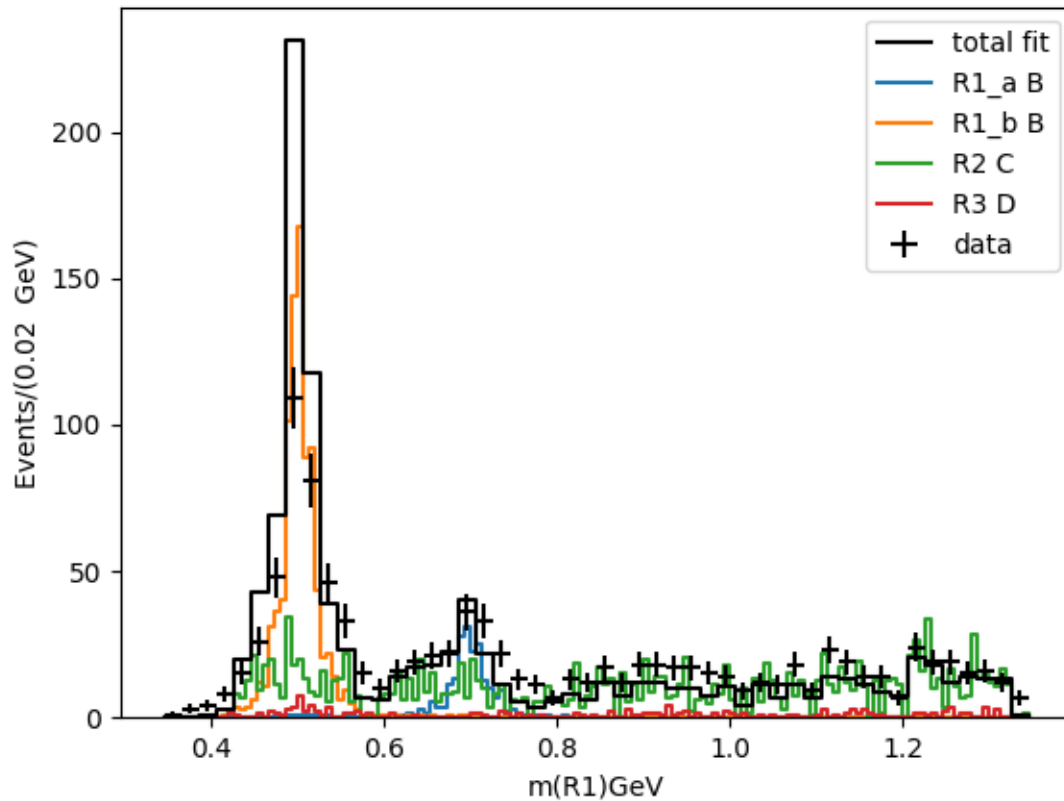
```

Ploting all partial waves is simple.

```

76 plotter.plot_frame("m_R1")
77 plt.show()

```

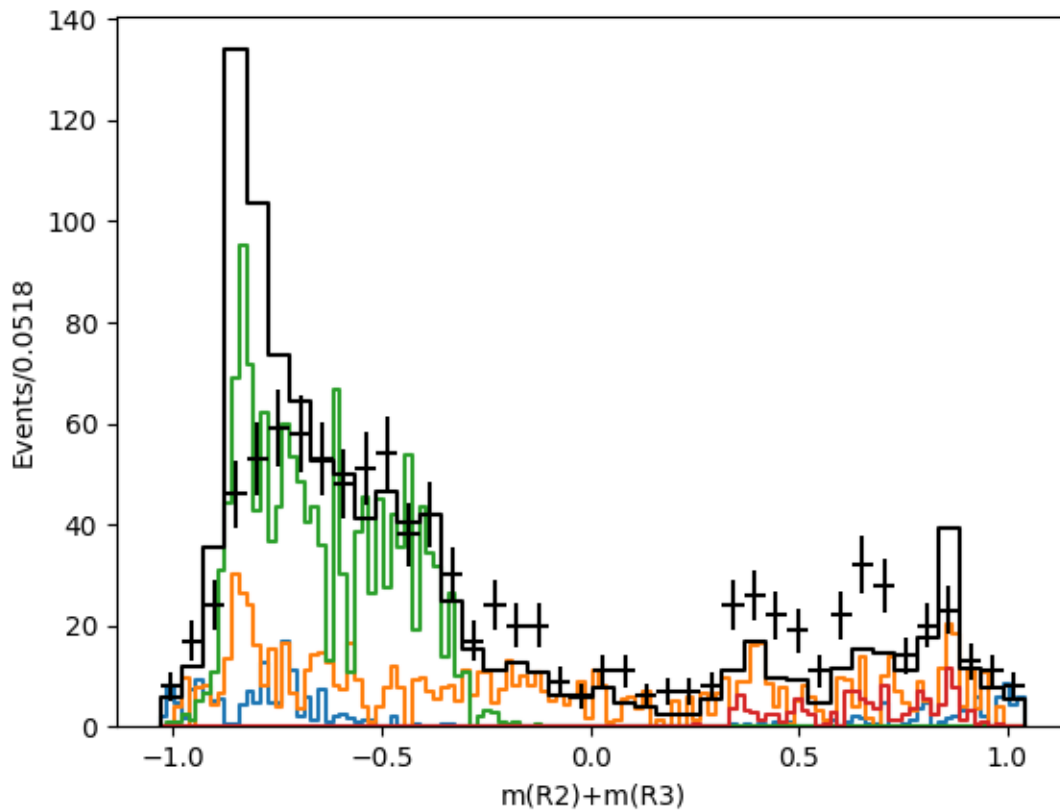


Also we can plot other variables in data

```

82 from tf_pwa.data import data_index
83
84 m2 = config.get_data_index("mass", "R2")
85 m3 = config.get_data_index("mass", "R3")
86
87
88 def f(data):
89     return data_index(data, m2) - data_index(data, m3)
90
91
92 plt.clf()
93 plotter.plot_var(f)
94 plt.xlabel("m(R2)+m(R3)")
95 plt.show()

```



There are 3 main parts in a Plotter

1. PlotAllData: datasets with weights There is three level: (1). idx: Datasets for combine fit (2). type: data, mc, or bg (3). observations and weights: weights are used for partial wave
2. Frame: function to get obseervations It is similar to RooFit's Frame.
3. Styles: Plot style for differencnt componets

The plot process is as follow:

1. Plotter.plot_item, extra_plot_item, and hidden_plot_item provide the list of histograms for plotting.
2. Loop over all data to get the observations though frame.
3. Frame provide the binning, weights from datas. Their combination is histogram
4. Plot on the axis with style

Total running time of the script: (0 minutes 6.720 seconds)

8.3 Particle and amplitude

Amplitude = DecayGroup + Variable

We will use following parameters for a toy model

```

11 from tf_pwa.amp import DecayChain, DecayGroup, get_decay, get_particle
12
13 resonances = {
14     "R0": {"J": 0, "P": 1, "mass": 1.0, "width": 0.07},
15     "R1": {"J": 0, "P": 1, "mass": 1.0, "width": 0.07},
16     "R2": {"J": 1, "P": -1, "mass": 1.225, "width": 0.08},
17 }
18
19 a, b, c, d = [get_particle(i, J=0, P=-1) for i in "ABCD"]
20 r1, r2, r3 = [get_particle(i, **resonances[i]) for i in resonances.keys()]
21
22
23 decay_group = DecayGroup(
24     [
25         DecayChain([get_decay(a, [r1, c]), get_decay(r1, [b, d])]),
26         DecayChain([get_decay(a, [r2, b]), get_decay(r2, [c, d])]),
27         DecayChain([get_decay(a, [r3, b]), get_decay(r3, [c, d])]),
28     ]
29 )

```

The above parts can be represented as config.yml used by ConfigLoader.

We can get AmplitudeModel from decay_group and a optional Variables Manager. It has parameters, so we can get and set parameters for the amplitude model

```

36 from tf_pwa.amp import AmplitudeModel
37 from tf_pwa.variable import VarsManager
38
39 vm = VarsManager()
40 amp = AmplitudeModel(decay_group, vm=vm)
41
42 print(amp.get_params())
43 amp.set_params(
44     {
45         "A->R0.CR0->B.D_total_0r": 1.0,
46         "A->R1.BR1->C.D_total_0r": 1.0,
47         "A->R2.BR2->C.D_total_0r": 7.0,
48     }
49 )

```

```

{'R0_mass': 1.0, 'R0_width': 0.07, 'R1_mass': 1.0, 'R1_width': 0.07, 'R2_mass': 1.225,
 → 'R2_width': 0.08, 'A->R0.CR0->B.D_total_0r': 1.5852381560068682, 'A->R0.CR0->B.D_total_
 → 0i': 1.690329569320169, 'A->R0.C_g_ls_0r': 1.0, 'A->R0.C_g_ls_0i': 0.0, 'R0->B.D_g_ls_
 → 0r': 1.0, 'R0->B.D_g_ls_0i': 0.0, 'A->R1.BR1->C.D_total_0r': 0.9942309682774577, 'A->
 → R1.BR1->C.D_total_0i': -2.2265736366054854, 'A->R1.B_g_ls_0r': 1.0, 'A->R1.B_g_ls_0i': 0.
 → 0.0, 'R1->C.D_g_ls_0r': 1.0, 'R1->C.D_g_ls_0i': 0.0, 'A->R2.BR2->C.D_total_0r': 0.
 → 7012398585867645, 'A->R2.BR2->C.D_total_0i': 2.0884932274966364, 'A->R2.B_g_ls_0r': 1.
 → 0, 'A->R2.B_g_ls_0i': 0.0, 'R2->C.D_g_ls_0r': 1.0, 'R2->C.D_g_ls_0i': 0.0}

```

For the calculation, we generate some phase space data.

```

54 from tf_pwa.phasespace import PhaseSpaceGenerator
55
56 m_A, m_B, m_C, m_D = 1.8, 0.18, 0.18, 0.18
57 p1, p2, p3 = PhaseSpaceGenerator(m_A, [m_B, m_C, m_D]).generate(1000000)

```

and the calculate helicity angle from the data

```

62 from tf_pwa.cal_angle import cal_angle_from_momentum
63
64 data = cal_angle_from_momentum({b: p1, c: p2, d: p3}, decay_group)

```

we can index mass from data as

```

69 from tf_pwa.data import data_index
70
71 m_bd = data_index(data, ("particle", "(B, D)", "m"))
72 # m_bc = data_index(data, ("particle", "(B, C)", "m"))
73 m_cd = data_index(data, ("particle", "(C, D)", "m"))

```

Note: If DecayGroup do not include resonant of (B, C), the data will not include its mass too. We can use different DecayGroup for cal_angle and AmplitudeModel when they have the same initial and final particle.

The amplitude square is calculated by amplitude model simply as

```

83 amp_s2 = amp(data)

```

```

/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/checkouts/stable/docs/./tf_pwa/
↪amp/core.py:803: UserWarning: no mass for particle A, set it to 1.800000022965964
  warnings.warn(
/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/checkouts/stable/docs/./tf_pwa/
↪amp/core.py:803: UserWarning: no mass for particle C, set it to 0.17999999999999977
  warnings.warn(
/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/checkouts/stable/docs/./tf_pwa/
↪amp/core.py:803: UserWarning: no mass for particle B, set it to 0.17999999999999977
  warnings.warn(
/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/checkouts/stable/docs/./tf_pwa/
↪amp/core.py:803: UserWarning: no mass for particle D, set it to 0.17999999999999977
  warnings.warn(

```

Now by using matplotlib we can get the Dalitz plot as

```

88 import matplotlib.pyplot as plt
89
90 plt.clf()
91 plt.hist2d(
92     m_bd.numpy() ** 2,
93     m_cd.numpy() ** 2,
94     weights=amp_s2.numpy(),
95     bins=60,
96     cmin=1,

```

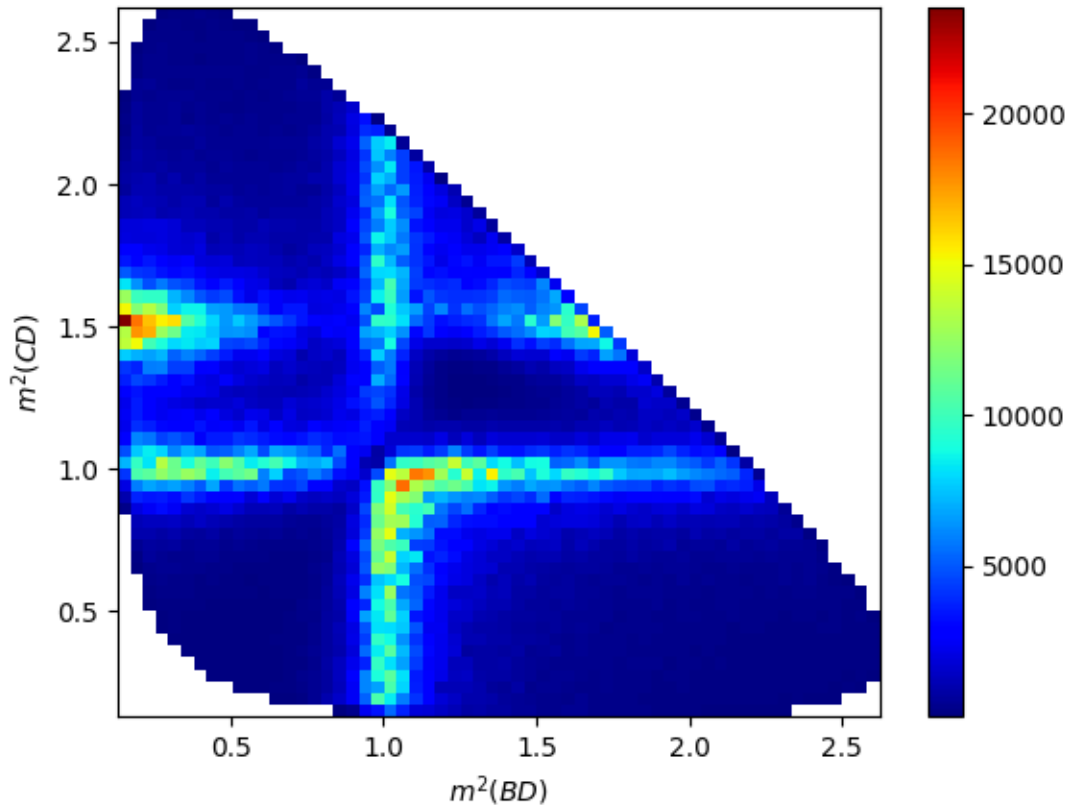
(continues on next page)

(continued from previous page)

```

97     cmap="jet",
98 )
99 plt.colorbar()
100 plt.xlabel("$m^2(BD)$")
101 plt.ylabel("$m^2(CD)$")
102 plt.show()

```



Total running time of the script: (0 minutes 4.831 seconds)

8.4 Examples for error propagation

Hear we use the same config in `particle_amplitude.py`

```

9  config_str = """
10 decay:
11   A:
12     - [R1, B]
13     - [R2, C]
14     - [R3, D]
15   R1: [C, D]
16   R2: [B, D]

```

(continues on next page)

(continued from previous page)

```

17     R3: [B, C]
18
19     particle:
20         $top:
21             A: { mass: 1.86, J: 0, P: -1}
22         $finals:
23             B: { mass: 0.494, J: 0, P: -1}
24             C: { mass: 0.139, J: 0, P: -1}
25             D: { mass: 0.139, J: 0, P: -1}
26         R1: [ R1_a, R1_b ]
27         R1_a: { mass: 0.7, width: 0.05, J: 1, P: -1}
28         R1_b: { mass: 0.5, width: 0.05, J: 0, P: +1}
29         R2: { mass: 0.824, width: 0.05, J: 0, P: +1}
30         R3: { mass: 0.824, width: 0.05, J: 0, P: +1}
31
32     """"
33
34     import matplotlib.pyplot as plt
35     import yaml
36
37     from tf_pwa.config_loader import ConfigLoader
38     from tf_pwa.histogram import Hist1D
39
40     config = ConfigLoader(yaml.full_load(config_str))
41     input_params = {
42         "A->R1_a.BR1_a->C.D_total_0r": 6.0,
43         "A->R1_b.BR1_b->C.D_total_0r": 1.0,
44         "A->R2.CR2->B.D_total_0r": 2.0,
45         "A->R3.DR3->B.C_total_0r": 1.0,
46     }
47     config.set_params(input_params)
48
49     data = config.generate_toy(1000)
50     phsp = config.generate_phsp(10000)

```

```

9.0%[>-----] 0.92/10.23s eff: 90.000000%
107.6%[>] 2.42/2.25s eff: 7.440720%
100.0%[] 2.42/2.42s eff: 7.338512%

```

After we calculated the parameters error, we will have an error matrix `config.inv_he` (using the inverse hessian). It is possible to save such matrix directly by `numpy.save` and to load it by `numpy.load`.

```

57     config.get_params_error(data=[data], phsp=[phsp])

```

Using Model

Time for calculating errors: 1.2068400382995605

hesse_error: [0.06625563845900341, 0.10777631290934468, 0.13143216360361248, 0.
 ↪ 0.09782302392447259, 0.08296139271348224, 0.12369480424462445]

```

{'A->R1_b.BR1_b->C.D_total_0r': 0.06625563845900341, 'A->R1_b.BR1_b->C.D_total_0i': 0.
↪ 10777631290934468, 'A->R2.CR2->B.D_total_0r': 0.13143216360361248, 'A->R2.CR2->B.D_
↪ total_0i': 0.09782302392447259, 'A->R3.DR3->B.C_total_0r': 0.08296139271348224, 'A->R3.

```

(continues on next page)

(continued from previous page)

```
→DR3->B.C_total_0i': 0.12369480424462445}
```

We can use the following method to profamance the error propagation

$$\sigma_f = \sqrt{\frac{\partial f}{\partial x_i} V_{ij} \frac{\partial f}{\partial x_j}}$$

by adding some calculation here. We need to use tensorflow functions instead of those of `math` or `numpy`.

```
68 import tensorflow as tf
69
70 with config.params_trans() as pt:
71     a2_r = pt["A->R2.CR2->B.D_total_0r"]
72     a2_i = pt["A->R2.CR2->B.D_total_0r"]
73     a2_x = a2_r * tf.cos(a2_i)
```

And then we can calculate the error we needed as

```
78 print(a2_x.numpy(), pt.get_error(a2_x).numpy())
```

```
-0.8322936730942848 0.2937169354379835
```

We can also calculate some more complex examples, such as the ratio in mass range (0.75, 0.85) over full phace space. Even further, we can get the error of error in the meaning of error propagation.

```
84 from tf_pwa.data import data_mask
85
86 m_R2 = phsp.get_mass("(B, D)")
87 cut_cond = (m_R2 < 0.85) & (m_R2 > 0.75)
88
89 amp = config.get_amplitude()
90
91 with config.params_trans() as pt1:
92     with config.params_trans() as pt:
93         int_mc = tf.reduce_sum(amp(phsp))
94         cut_phsp = data_mask(phsp, cut_cond)
95         cut_int_mc = tf.reduce_sum(amp(cut_phsp))
96         ratio = cut_int_mc / int_mc
97         error = pt.get_error(ratio)
98
99 print(ratio.numpy(), "+/-", error.numpy())
100 print(error.numpy(), "+/-", pt1.get_error(error).numpy())
```

```
0.38197185921153415 +/- 0.01198798472005041
0.01198798472005041 +/- 0.0008233112942379798
```

Total running time of the script: (0 minutes 5.788 seconds)

8.5 Examples for config.yml file

Configuration file config.yml use YAML (<https://yaml.org>) format to describe decay process.

The main parts of config.yml is decay and particle.

The decay part describe the particle (or an id of a list of particle) decay into which particles, it can be a list of a list of list. A list means that there is only one decay mode, A list of list is the list of possible decay mode. The list item can be the particle name (or a dict to describe the decay parameters). All name should appear in particle part.

The particle part describe the parameters of particles. There are two special parts \$stop and \$finals describe the top and finals particles. The other parts are lists of particle name or dicts of particle parameters. The list is the same type particle in decay part. The dict is the parameters of the particle name.

```

22 config_str = """
23
24 decay:
25     A:
26         - [R1, B]
27         - [R2, C]
28         - [R3, D]
29     R1: [C, D]
30     R2: [B, D]
31     R3: [B, C]
32
33 particle:
34     $stop:
35         A: { mass: 1.86, J: 0, P: -1}
36     $finals:
37         B: { mass: 0.494, J: 0, P: -1}
38         C: { mass: 0.139, J: 0, P: -1}
39         D: { mass: 0.139, J: 0, P: -1}
40     R1: [ R1_a, R1_b ]
41     R1_a: { mass: 0.7, width: 0.05, J: 1, P: -1}
42     R1_b: { mass: 0.5, width: 0.05, J: 0, P: +1}
43     R2: { mass: 0.824, width: 0.05, J: 0, P: +1}
44     R3: { mass: 0.824, width: 0.05, J: 0, P: +1}
45
46 """

```

The config file can be loaded by yaml library.

```

52 import matplotlib.pyplot as plt
53 import yaml
54
55 from tf_pwa.config_loader import ConfigLoader
56 from tf_pwa.histogram import Hist1D

```

The simple way to create config is write it to config.yml file and then you can load it as config = ConfigLoader("config.yml"). Here we used config_str directly.

```

64 config = ConfigLoader(yaml.full_load(config_str))

```

We set parameters to a blance value. And we can generate some toy data and calculte the weights The full params can be found by print(config.get_params()).

```

71 input_params = {
72     "A->R1_a.BR1_a->C.D_total_0r": 6.0,
73     "A->R1_b.BR1_b->C.D_total_0r": 1.0,
74     "A->R2.CR2->B.D_total_0r": 2.0,
75     "A->R3.DR3->B.C_total_0r": 1.0,
76 }
77 config.set_params(input_params)

```

True

Here we generate some toy data and phsp mc to show the model

```

82 data = config.generate_toy(1000)
83 phsp = config.generate_phsp(10000)

```

```

7.9%[>-----] 0.91/11.55s eff: 90.000000%
98.6%[>] 2.76/2.80s eff: 6.541292%
99.7%[>] 3.61/3.62s eff: 5.906643%
100.3%[>] 4.42/4.41s eff: 5.880966%
100.0%[] 4.42/4.42s eff: 5.896864%

```

You can also fit the data fit to the data We can omit the args when written in config.yml

```

90 fit_result = config.fit([data], [phsp])
91 # After the fit, you can get the uncertainties as
92 err = config.get_params_error(fit_result, [data], [phsp])

```

Using Model

decay chains included:

```

[A->R1_a+B, R1_a->C+D] ls: ((1, 1),) ((1, 0),)
[A->R1_b+B, R1_b->C+D] ls: ((0, 0),) ((0, 0),)
[A->R2+C, R2->B+D] ls: ((0, 0),) ((0, 0),)
[A->R3+D, R3->B+C] ls: ((0, 0),) ((0, 0),)

```

initial parameters

```

{
  "R1_a_mass": 0.7,
  "R1_a_width": 0.05,
  "R1_b_mass": 0.5,
  "R1_b_width": 0.05,
  "R2_mass": 0.824,
  "R2_width": 0.05,
  "R3_mass": 0.824,
  "R3_width": 0.05,
  "A->R1_a.BR1_a->C.D_total_0r": 6.0,
  "A->R1_a.BR1_a->C.D_total_0i": 0.0,
  "A->R1_a.B_g_ls_0r": 1.0,
  "A->R1_a.B_g_ls_0i": 0.0,
  "R1_a->C.D_g_ls_0r": 1.0,
  "R1_a->C.D_g_ls_0i": 0.0,
  "A->R1_b.BR1_b->C.D_total_0r": 1.0,
  "A->R1_b.BR1_b->C.D_total_0i": -0.17036099828961904,

```

(continues on next page)

(continued from previous page)

```

"A->R1_b.B_g_ls_0r": 1.0,
"A->R1_b.B_g_ls_0i": 0.0,
"R1_b->C.D_g_ls_0r": 1.0,
"R1_b->C.D_g_ls_0i": 0.0,
"A->R2.CR2->B.D_total_0r": 2.0,
"A->R2.CR2->B.D_total_0i": -2.44846429485405,
"A->R2.C_g_ls_0r": 1.0,
"A->R2.C_g_ls_0i": 0.0,
"R2->B.D_g_ls_0r": 1.0,
"R2->B.D_g_ls_0i": 0.0,
"A->R3.DR3->B.C_total_0r": 1.0,
"A->R3.DR3->B.C_total_0i": -1.5299171823613154,
"A->R3.D_g_ls_0r": 1.0,
"A->R3.D_g_ls_0i": 0.0,
"R3->B.C_g_ls_0r": 1.0,
"R3->B.C_g_ls_0i": 0.0
}
initial NLL: tf.Tensor(-876.5550510935509, shape=(), dtype=float64)
nll_grad cost time: 0.4722561836242676
nll_grad cost time: 0.474503755569458
nll_grad cost time: 0.48148345947265625
tf.Tensor(-877.5403955489946, shape=(), dtype=float64)
nll_grad cost time: 0.4866471290588379
nll_grad cost time: 0.4884514808654785
tf.Tensor(-877.9356094551549, shape=(), dtype=float64)
nll_grad cost time: 0.47771215438842773
tf.Tensor(-878.4860877702886, shape=(), dtype=float64)
nll_grad cost time: 0.4797697067260742
nll_grad cost time: 0.4736459255218506
tf.Tensor(-878.7509883100593, shape=(), dtype=float64)
nll_grad cost time: 0.49033093452453613
tf.Tensor(-878.8244008216343, shape=(), dtype=float64)
nll_grad cost time: 0.49677062034606934
tf.Tensor(-878.9546735802714, shape=(), dtype=float64)
nll_grad cost time: 0.47876691818237305
tf.Tensor(-879.1570792905513, shape=(), dtype=float64)
nll_grad cost time: 0.4896061420440674
tf.Tensor(-879.4227001224172, shape=(), dtype=float64)
nll_grad cost time: 0.49108386039733887
tf.Tensor(-879.4463424397973, shape=(), dtype=float64)
nll_grad cost time: 0.49104857444763184
tf.Tensor(-879.4464152430883, shape=(), dtype=float64)
nll_grad cost time: 0.4891695976257324
tf.Tensor(-879.4464153398749, shape=(), dtype=float64)
Optimization terminated successfully.
    Current function value: -879.446415
    Iterations: 11
    Function evaluations: 15
    Gradient evaluations: 15
message: Optimization terminated successfully.
success: True
status: 0

```

(continues on next page)

(continued from previous page)

```

fun: -879.4464153398749
  x: [ 9.489e-01 -9.681e-02  1.885e+00 -2.304e+00  8.685e-01
      -1.461e+00]
nit: 11
jac: [-1.631e-04  6.923e-05  4.877e-05  1.047e-04 -2.877e-05
      2.223e-06]
hess_inv: [[ 3.302e-03  4.340e-04 ...  2.010e-03  1.184e-03]
           [ 4.340e-04  1.302e-02 ...  6.632e-04  6.411e-03]
           ...
           [ 2.010e-03  6.632e-04 ...  4.321e-03  2.374e-04]
           [ 1.184e-03  6.411e-03 ...  2.374e-04  1.835e-02]]
nfev: 15
njev: 15
fit cost time: 7.779327869415283
Using Model
Time for calculating errors: 1.1640501022338867
hesse_error: [0.056765147154824135, 0.11184095287604047, 0.1121442870963205, 0.
↪ 10192939584029746, 0.06562663234319398, 0.13551414787699745]

```

we can see that thre fit results consistant with inputs, the first one is fixed.

```

97  for var in input_params:
98      print(
99          f"in: {input_params[var]} => out: {fit_result.params[var]} +/- {err.get(var, 0.
↪ )}"
100      )

```

```

in: 6.0 => out: 6.0 +/- 0.0
in: 1.0 => out: 0.9489399180798221 +/- 0.056765147154824135
in: 2.0 => out: 1.884781669098602 +/- 0.1121442870963205
in: 1.0 => out: 0.8685280835804645 +/- 0.06562663234319398

```

We can use the amplitude to plot the fit results

```

105 amp = config.get_amplitude()

```

This is the total $|A|^2$

```

110 weight = amp(phsp)

```

This is the $|A_i|^2$ for each decay chain

```

115 partial_weight = amp.partial_weight(phsp)

```

We can plot the data, Hist1D include some plot method base on matplotlib.

```

120 data_hist = Hist1D.histogram(
121     data.get_mass("(C, D)"), bins=60, range=(0.25, 1.45)
122 )

```

Read the mass var from phsp.

```

127 mass_phsp = phsp.get_mass("(C, D)")

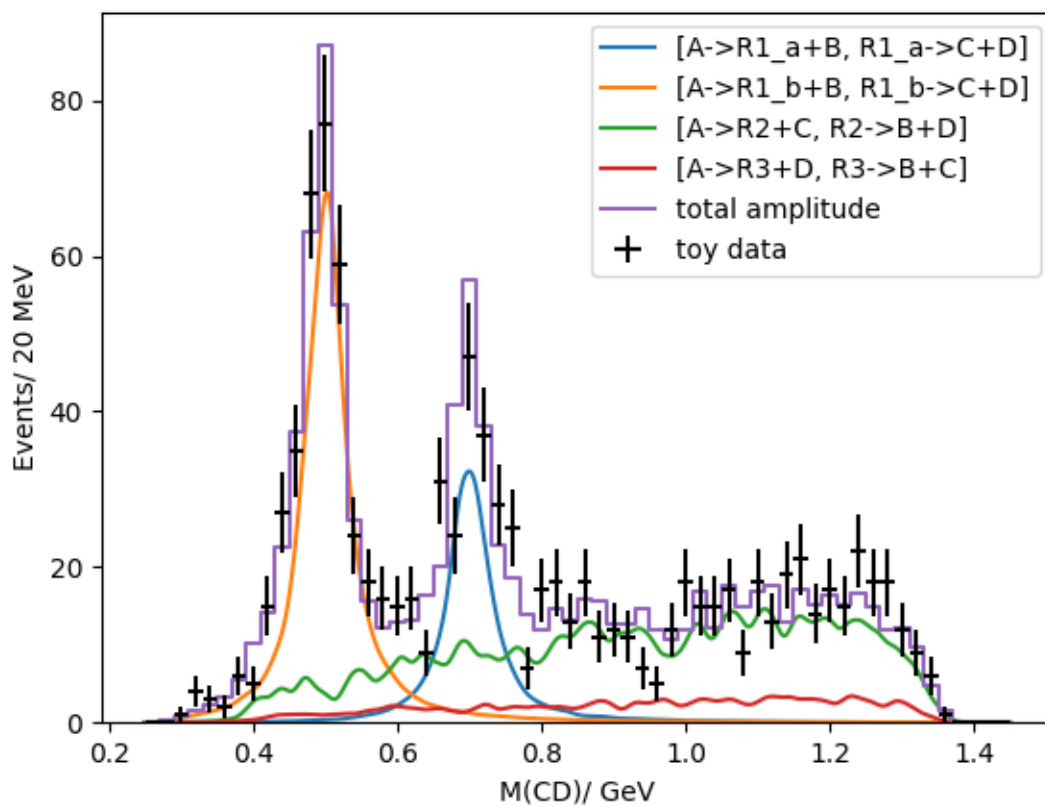
```

For helicity angle it can be read as `phsp.get_angle("C, D)", "C")` The first arg is used to determine the decay chain. (decay chain include all particles) The second arg is used to determine the angle in decay chain. The return value is a dict as `{"alpha": phi, "beta": theta, "gamma": 0}`

```
136 phsp_hist = Hist1D.histogram(  
137     mass_phsp, weights=weight, bins=60, range=(0.25, 1.45)  
138 )  
139 scale = phsp_hist.scale_to(data_hist)  
140  
141 pw_hist = []  
142 for w in partial_weight:  
143     # here we used more bins for a smooth plot  
144     hist = Hist1D.histogram(  
145         mass_phsp, weights=w, bins=60 * 2, range=(0.25, 1.45)  
146     )  
147     pw_hist.append(scale * hist * 2)
```

Then we can plot the histogram into matplotlib

```
152 for hist, dec in zip(pw_hist, config.get_decay()):  
153     hist.draw_kde(label=str(dec))  
154 phsp_hist.draw(label="total amplitude")  
155 data_hist.draw_error(label="toy data", color="black")  
156  
157 plt.legend()  
158 plt.ylim((0, None))  
159 plt.xlabel("M(CD)/ GeV")  
160 plt.ylabel("Events/ 20 MeV")  
161 plt.show()
```

Total running time of the script: (0 minutes 15.531 seconds)

SETUP FOR DEVELOPER ENVERIMENT

Note: Before the developing, creating a standalone enveriment is recomanded (see <https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands> for more).

The main steps are similar as normal install, only two extra things need to be done.

The first things is writing tests, and tests your code. We use pytest (<https://docs.pytest.org/en/stable/>) framework, You should install it.

```
conda install pytest pytest-cov pytest-benchmark
```

The other things is pre-commit. it need for developing.

1. You can install pre-commit as

```
conda install pre-commit
```

and

2. then enable pre-commit in the source dir

```
conda install pylint # local dependences  
pre-commit install
```

You can check if pre-commit is working well by running

```
pre-commit run -a
```

It may take some time to install required package.

Note: If there are some GLIBC_XXX errors at this step, you can try to install node.js.

Note: For developer using editor with formatter, you should be careful for the options.

The following are all commands needed

```
# create environment  
conda create -n tfpwa2 python=3.7 -y  
conda activate tfpwa2
```

(continues on next page)

(continued from previous page)

```
# install tf-pwa
conda install --file requirements-min.txt -y
python -m pip install -e . --no-deps
# install pytest
conda install pytest pytest-cov -y
# install pylint local
conda install pylint
# install pre-commit
conda install pre-commit -c conda-forge -y
pre-commit install
```

10.1 tf_pwa

Partial Wave Analysis program using Tensorflow

Submodules and Subpackages

10.1.1 amp

Basic Amplitude Calculations. A partial wave analysis process has following structure:

DecayGroup: addition (+)

DecayChain: multiplication (x)
Decay, Particle(Propagator)

Submodules and Subpackages

Kmatrix

B1($l, q, q0, d=3$)

Fb($l, q, d=3$)

KMatrix_single($n_pole, m1, m2, l, d=3, bkg=0, Kb=0$)

class KmatrixSingleChannelParticle(*args, **kwargs)

Bases: *Particle*

K matrix model for single channel multi pole.

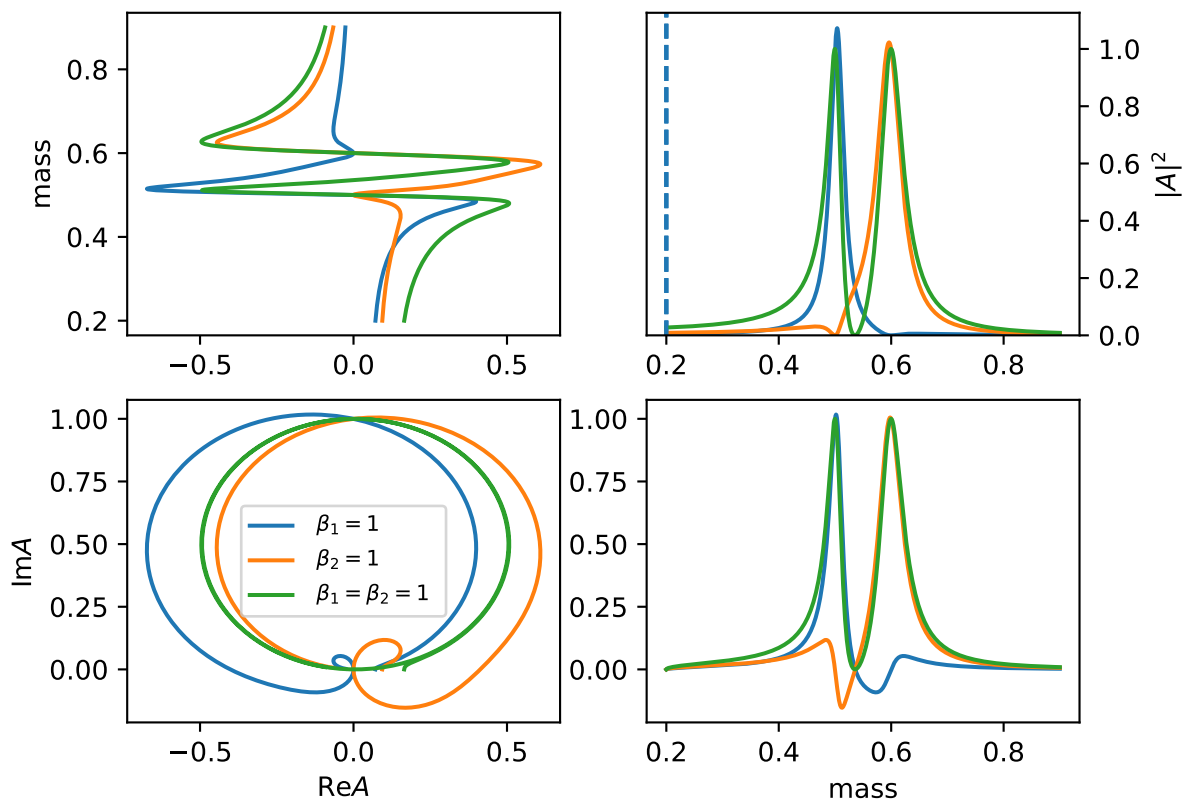
$$K = \sum_i \frac{m_i \Gamma_i(m)}{m_i^2 - m^2}$$

$$P = \sum_i \frac{\beta_i m_0 \Gamma_0}{m_i^2 - m^2}$$

the barrier factor is included in gls

$$R(m) = (1 - iK)^{-1}P$$

required mass_list: [pole1, pole2] and width_list: [width1, width2].



```

    get_amp(*args, **kwargs)

    get_beta()

    get_gi()

    get_mi()

    init_params()

    model_name = 'KMatrixSingleChannel'

class KmatrixSplitLSParticle(*args, **kwargs)

```

Bases: [Particle](#)

K matrix model for single channel multi pole and the same channel with different (l, s) coupling.

$$K_{a,b} = \sum_i \frac{m_i \sqrt{\Gamma_{a,i}(m) \Gamma_{b,i}(m)}}{m_i^2 - m^2}$$

$$P_b = \sum_i \frac{\beta_i m_0 \Gamma_{b,i0}}{m_i^2 - m^2}$$

the barrier factor is included in gls

$$R(m) = (1 - iK)^{-1}P$$

```

    get_amp(*args, **kwargs)

    get_beta()

    get_gi()

    get_gi_frac()

    get_ls_amp(m)

    get_mi()

    init_params()

    model_name = 'KMatrixSplitLS'

class ParticleDecayLSKmatrix(*args, **kwargs)
    Bases: HelicityDecay

    get_ls_amp(data, data_p, **kwargs)

    init_params()

    model_name = 'LS-decay-Kmatrix'

get_relative_p(m0, m1, m2)

opt_lambdify(args, expr, **kwargs)

```

amp

```
class AbsPDF(*args, name="", vm=None, polar=None, use_tf_function=False, no_id_cached=False,
             jit_compile=False, **kwargs)
```

```
    Bases: object
```

```
    cached_available()
```

```
    get_params(trainable_only=False)
```

```
    mask_params(var)
```

```
    set_params(var)
```

```
    temp_params(var)
```

```
    property trainable_variables
```

```
    property variables
```

```
class AmplitudeModel(decay_group, **kwargs)
```

```
    Bases: BaseAmplitudeModel
```

```
    partial_weight(data, combine=None)
```

```
class BaseAmplitudeModel(decay_group, **kwargs)
```

```
    Bases: AbsPDF
```

```
    cache_data(data, split=None, batch=None)
```

```
    cached_available()
```

```
    chains_particle()
```

```
    factor_iteration(deep=2)
```

```
    init_params(name="")
```

```
    partial_weight(data, combine=None)
```

```
    partial_weight_interference(data)
```

```
    pdf(data)
```

```
    set_used_chains(used_chains)
```

```
    set_used_res(res)
```

```
    temp_total_gls_one()
```

```
    temp_used_res(res)
```

```
class CachedAmpAmplitudeModel(decay_group, **kwargs)
```

```
    Bases: BaseAmplitudeModel
```

```
    pdf(data)
```

```
class CachedShapeAmplitudeModel(*args, **kwargs)
```

```
    Bases: BaseAmplitudeModel
```



```

    get_cached_shape_idx()

    pdf(data)

class FactorAmplitudeModel(*args, **kwargs)
    Bases: BaseAmplitudeModel

    get_amp_list(data)

    get_amp_list_part(data)

    pdf(data)

class P4DirectlyAmplitudeModel(decay_group, **kwargs)
    Bases: BaseAmplitudeModel

    cal_angle(p4)

    pdf(data)

create_amplitude(decay_group, **kwargs)

register_amp_model(name=None, f=None)
    register a data mode

    Params name
        mode name used in configuration

    Params f
        Data Mode class

base

Basic amplitude model

class HelicityDecayCPV(*args, has_barrier_factor=True, l_list=None, barrier_factor_mass=False,
                        has_ql=True, has_bprime=True, aligned=False, allow_cc=True, ls_list=None,
                        barrier_factor_norm=False, params_polar=None, below_threshold=False,
                        force_min_l=False, params_head=None, no_q0=False, helicity_inner_full=False,
                        ls_selector=None, **kwargs)

    Bases: HelicityDecay

    decay model for CPV

    get_g_ls(charge=1)

    get_ls_amp(data, data_p, **kwargs)

    init_params()

    model_name = 'gls-cpv'

class HelicityDecayNP(*args, has_barrier_factor=True, l_list=None, barrier_factor_mass=False,
                       has_ql=True, has_bprime=True, aligned=False, allow_cc=True, ls_list=None,
                       barrier_factor_norm=False, params_polar=None, below_threshold=False,
                       force_min_l=False, params_head=None, no_q0=False, helicity_inner_full=False,
                       ls_selector=None, **kwargs)

```

Bases: [HelicityDecay](#)

Full helicity amplitude

$$A = H_{m_1, m_2} D_{m_0, m_1 - m_2}^{J_0*}(\varphi, \theta, 0)$$

fit parameters is H_{m_1, m_2} .

fix_unused_h()

get_H()

get_H_zero_mask()

get_factor()

get_factor_H(data=None, data_p=None, **kwargs)

get_helicity_amp(data=None, data_p=None, **kwargs)

get_ls_amp(data, data_p, **kwargs)

get_zero_index()

init_params()

model_name = 'helicity_full'

```
class HelicityDecayNPbf(*args, has_barrier_factor=True, l_list=None, barrier_factor_mass=False,
    has_ql=True, has_bprime=True, aligned=False, allow_cc=True, ls_list=None,
    barrier_factor_norm=False, params_polar=None, below_threshold=False,
    force_min_l=False, params_head=None, no_q0=False, helicity_inner_full=False,
    ls_selector=None, **kwargs)
```

Bases: [HelicityDecayNP](#)

get_H_barrier_factor(data, data_p, **kwargs)

get_helicity_amp(data, data_p, **kwargs)

get_ls_amp(data, data_p, **kwargs)

init_params()

model_name = 'helicity_full-bf'

```
class HelicityDecayP(*args, has_barrier_factor=True, l_list=None, barrier_factor_mass=False, has_ql=True,
    has_bprime=True, aligned=False, allow_cc=True, ls_list=None,
    barrier_factor_norm=False, params_polar=None, below_threshold=False,
    force_min_l=False, params_head=None, no_q0=False, helicity_inner_full=False,
    ls_selector=None, **kwargs)
```

Bases: [HelicityDecayNP](#)

$$H_{-m_1, -m_2} = P_0 P_1 P_2 (-1)^{J_1 + J_2 - J_0} H_{m_1, m_2}$$

get_helicity_amp(data, data_p, **kwargs)

```

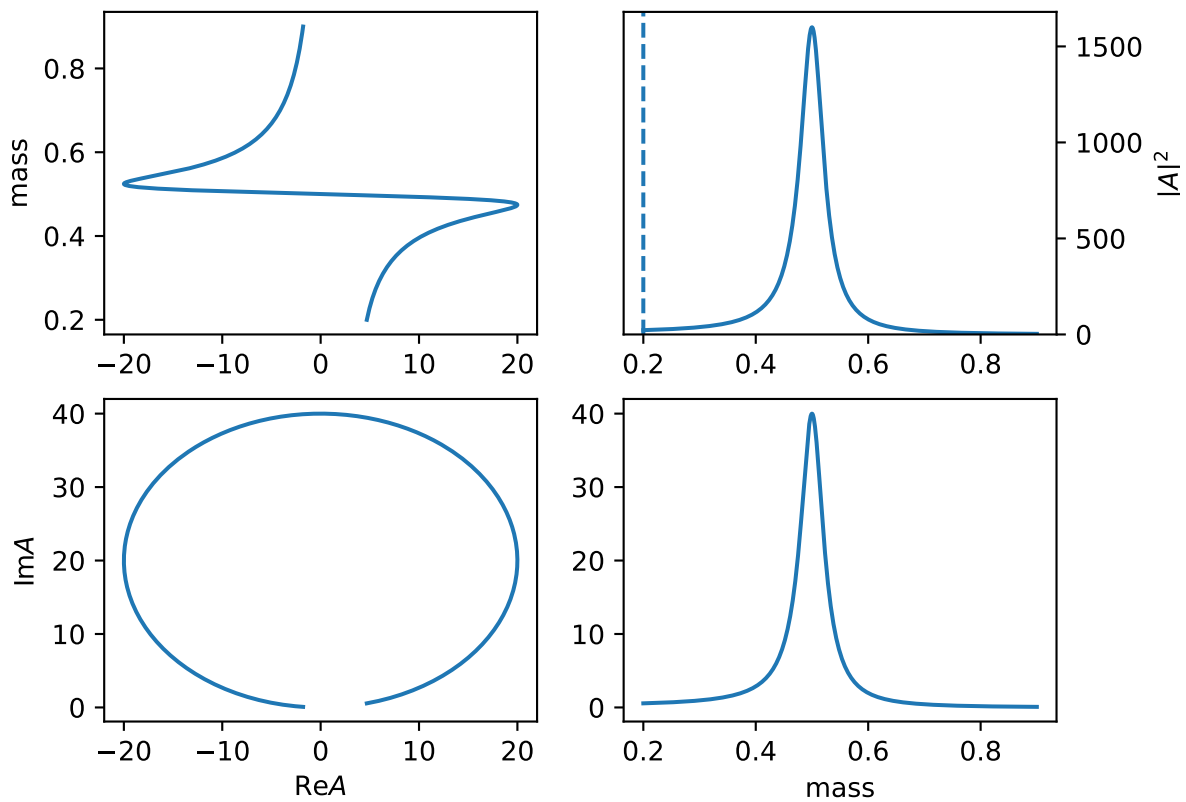
init_params()

model_name = 'helicity_parity'

class ParticleBW(*args, running_width=True, bw_l=None, width_norm=False, params_head=None, **kwargs)
    Bases: Particle

```

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0}$$



```

get_amp(data, _data_c=None, **kwargs)

get_num_var()

get_sympy_var()

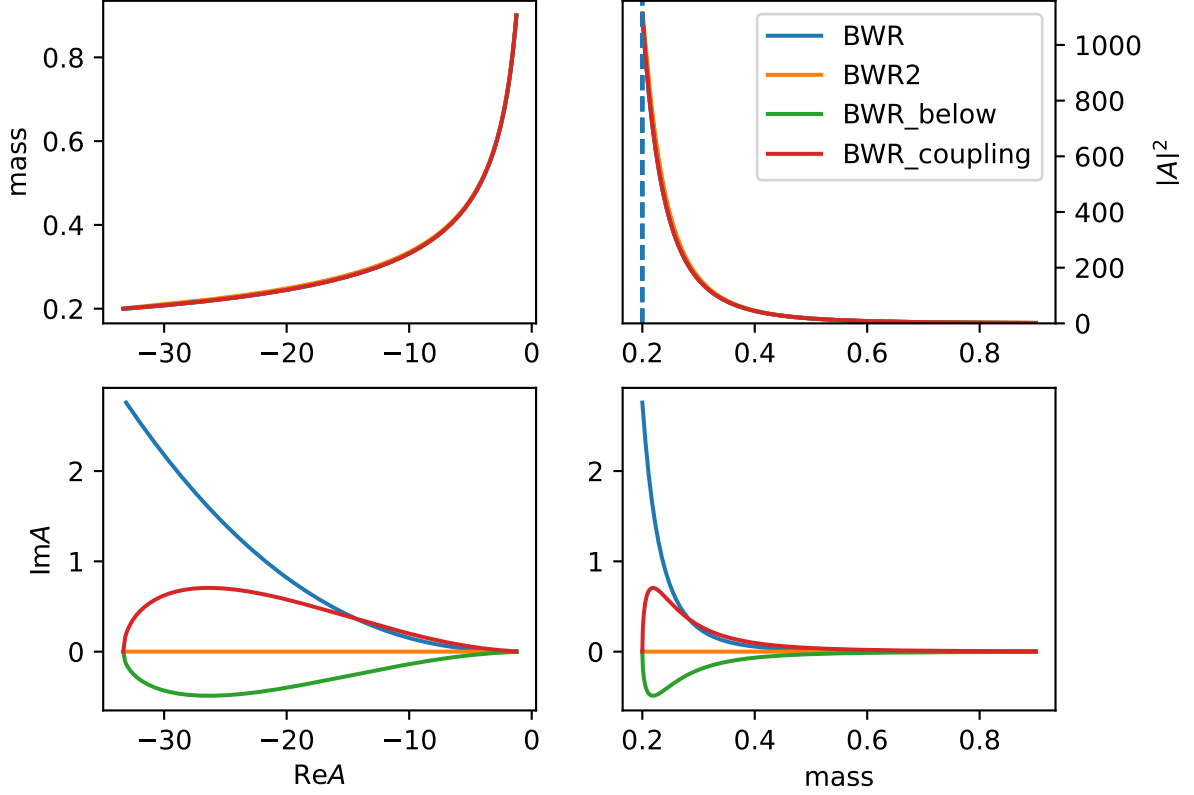
model_name = 'BW'

class ParticleBWR2(*args, running_width=True, bw_l=None, width_norm=False, params_head=None,
    **kwargs)
    Bases: Particle

```

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

The difference of BWR, BWR2 is the behavior when mass is below the threshold ($m_0 = 0.1 < 0.1 + 0.1 = m_1 + m_2$).



```
get_amp(data, data_c, **kwargs)
```

```
model_name = 'BWR2'
```

```
class ParticleBWRBelowThreshold(*args, running_width=True, bw_l=None, width_norm=False,
                                params_head=None, **kwargs)
```

Bases: [Particle](#)

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

```
get_amp(data, data_c, **kwargs)
```

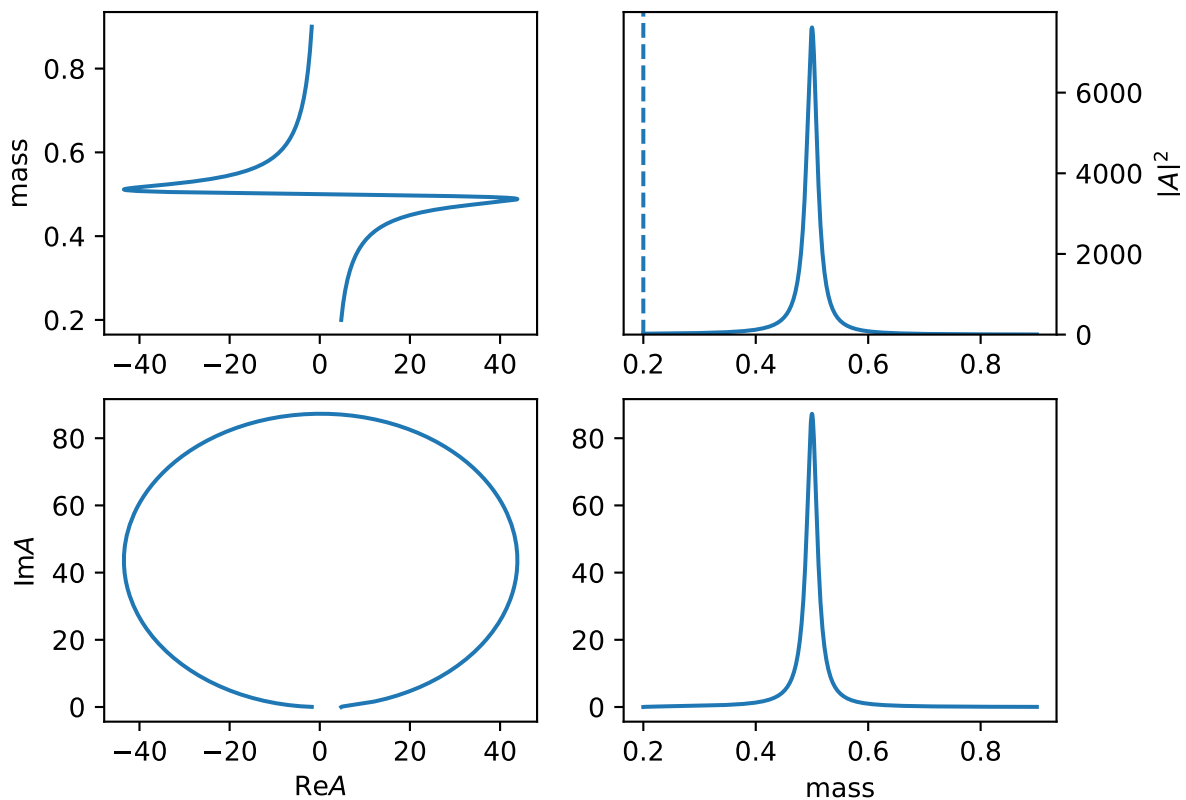
```
model_name = 'BWR_below'
```

```
class ParticleBWRCoupling(*args, running_width=True, bw_l=None, width_norm=False,
                           params_head=None, **kwargs)
```

Bases: [Particle](#)

Force $q_0 = 1/d$ to avoid below threshold condition for BWR model, and remove other constant parts, then the Γ_0 is coupling parameters.

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0 \frac{q}{m} q^{2l} B_L'^2(q, 1/d, d)}$$



```
get_amp(data, data_c, **kwargs)
```

```
get_sympy_dom(m, m0, g0, m1=None, m2=None, sheet=0)
```

```
model_name = 'BWR_coupling'
```

```
class ParticleBWR_normal(*args, running_width=True, bw_l=None, width_norm=False, params_head=None,
                        **kwargs)
```

Bases: [Particle](#)

$$R(m) = \frac{\sqrt{m_0 \Gamma(m)}}{m_0^2 - m^2 - i m_0 \Gamma(m)}$$

```
get_amp(data, data_c, **kwargs)
```

```
model_name = 'BWR_normal'
```

```
class ParticleDecay(*args, has_barrier_factor=True, l_list=None, barrier_factor_mass=False, has_q1=True,
                    has_bprime=True, aligned=False, allow_cc=True, ls_list=None,
                    barrier_factor_norm=False, params_polar=None, below_threshold=False,
                    force_min_l=False, params_head=None, no_q0=False, helicity_inner_full=False,
                    ls_selector=None, **kwargs)
```

Bases: *HelicityDecay*

get_ls_amp(data, data_p, **kwargs)

model_name = 'particle-decay'

class ParticleExp(*args, running_width=True, bw_l=None, width_norm=False, params_head=None, **kwargs)

Bases: *Particle*

$$R(m) = e^{-|a|m}$$

get_amp(data, _data_c=None, **kwargs)

init_params()

model_name = 'exp'

class ParticleExpCom(*args, running_width=True, bw_l=None, width_norm=False, params_head=None, **kwargs)

Bases: *Particle*

$$R(m) = e^{-(a+ib)m^2}$$

lineshape when $a = 1.0, b = 10$.

get_amp(data, _data_c=None, **kwargs)

init_params()

model_name = 'exp_com'

class ParticleGS(*args, **kwargs)

Bases: *Particle*

Gounaris G.J., Sakurai J.J., Phys. Rev. Lett., 21 (1968), pp. 244-247

c_daug2Mass: mass for daughter particle 2 (π^+) 0.13957039

c_daug3Mass: mass for daughter particle 3 (π^0) 0.1349768

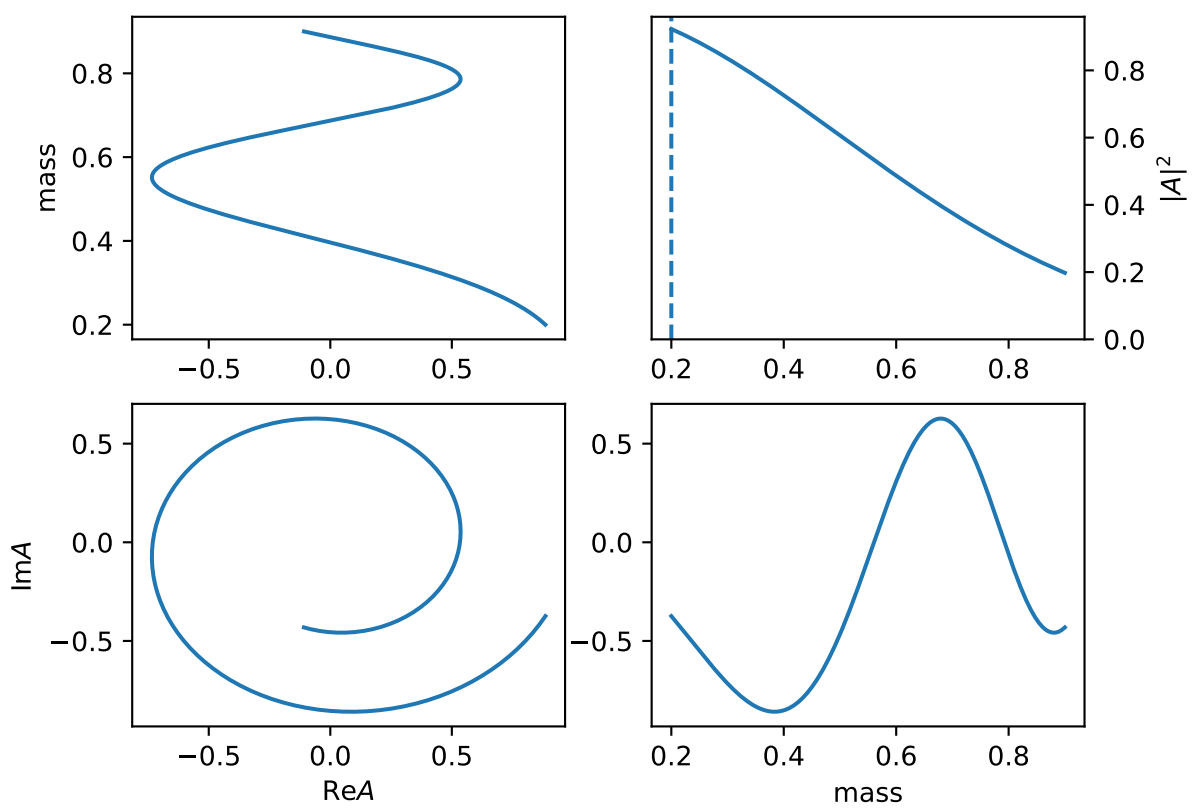
$$R(m) = \frac{1 + D\Gamma_0/m_0}{(m_0^2 - m^2) + f(m) - im_0\Gamma(m)}$$

$$f(m) = \Gamma_0 \frac{m_0^2}{q_0^3} \left[q^2 [h(m) - h(m_0)] + (m_0^2 - m^2) q_0^2 \frac{dh}{dm} \Big|_{m_0} \right]$$

$$h(m) = \frac{2}{\pi} \frac{q}{m} \ln \left(\frac{m + 2q}{2m_\pi} \right)$$

$$\frac{dh}{dm} \Big|_{m_0} = h(m_0) [(8q_0^2)^{-1} - (2m_0^2)^{-1}] + (2\pi m_0^2)^{-1}$$

$$D = \frac{f(0)}{\Gamma_0 m_0} = \frac{3}{\pi} \frac{m_\pi^2}{q_0^2} \ln \left(\frac{m_0 + 2q_0}{2m_\pi} \right) + \frac{m_0}{2\pi q_0} - \frac{m_\pi^2 m_0}{\pi q_0^3}$$



```
get_amp(data, data_c, **kwargs)

model_name = 'GS_rho'

class ParticleKmatrix(*args, running_width=True, bw_l=None, width_norm=False, params_head=None,
                      **kwargs)

    Bases: Particle

    get_amp(data, data_c=None, **kwargs)

    get_beta(m, **kwargs)

    init_params()

    model_name = 'Kmatrix'

class ParticleLass(*args, running_width=True, bw_l=None, width_norm=False, params_head=None,
                  **kwargs)

    Bases: Particle

    get_amp(data, data_c=None, **kwargs)
```

$$R(m) = \frac{m}{q \cot \delta_B - iq} + e^{2i\delta_B} \frac{m_0 \Gamma_0 \frac{m_0}{q_0}}{(m_0^2 - m^2) - im_0 \Gamma_0 \frac{q}{m} \frac{m_0}{q_0}}$$
$$\cot \delta_B = \frac{1}{aq} + \frac{1}{2}rq$$
$$e^{2i\delta_B} = \cos 2\delta_B + i \sin 2\delta_B = \frac{\cot^2 \delta_B - 1}{\cot^2 \delta_B + 1} + i \frac{2 \cot \delta_B}{\cot^2 \delta_B + 1}$$

```
    init_params()

    model_name = 'LASS'

class ParticleOne(*args, running_width=True, bw_l=None, width_norm=False, params_head=None,
                 **kwargs)

    Bases: Particle


```

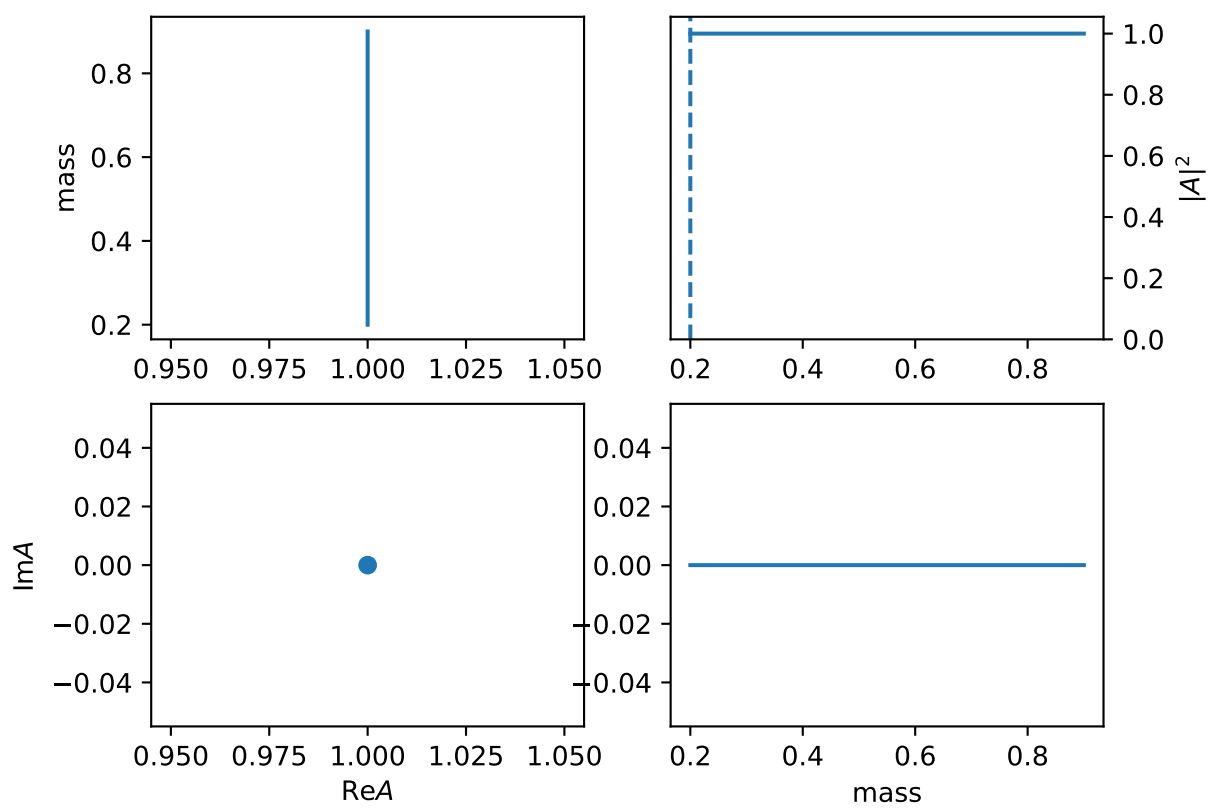
$$R(m) = 1$$

```
    get_amp(data, _data_c=None, **kwargs)

    init_params()

    model_name = 'one'

get_parity_term(j1, p1, j2, p2, j3, p3)
```

core

Basic Amplitude Calculations. A partial wave analysis process has following structure:

DecayGroup: addition (+)

DecayChain: multiplication (x)

Decay, Particle(Propagator)

class AmpBase

Bases: [object](#)

Base class for amplitude

add_var(*names, is_complex=False, shape=(), **kwargs*)

default add_var method

amp_shape()

get_factor_variable()

get_params_head()

get_var(*name*)

get_variable_name(*name=""*)

class AmpDecay(*core, outs, name=None, disable=False, p_break=False, c_break=True, curve_style=None, **kwargs*)

Bases: [Decay](#), [AmpBase](#)

base class for decay with amplitude

amp_index(*base_map*)

amp_shape()

get_params_head()

list_helicity_inner()

n_helicity_inner()

class AmpDecayChain(**args, is_cp=False, aligned=True, **kwargs*)

Bases: [DecayChain](#), [AmpBase](#)

get_params_head()

class AngSam3Decay(*core, outs, name=None, disable=False, p_break=False, c_break=True, curve_style=None, **kwargs*)

Bases: [AmpDecay](#), [AmpBase](#)

get_amp(*data, data_extra=None, **kwargs*)

init_params()

model_name = 'default'

```

class DecayChain(*args, is_cp=False, aligned=True, **kwargs)
    Bases: AmpDecayChain
    A list of Decay as a chain decay
    amp_index(base_map=None)
    amp_shape()
    factor_iteration(deep=1)
    get_all_factor()
    get_amp(data_c, data_p, all_data=None, base_map=None)
    get_amp_particle(data_p, data_c, all_data=None)
    get_amp_total(charge=1)
    get_angle_amp(data_c, data_p, all_data=None, base_map=None)
    get_base_map(base_map=None)
    get_cp_amp_total(charge=1)
    get_factor()
    get_factor_angle_amp(data_c, data_p, all_data=None, base_map=None)
    get_factor_variable()
    get_m_dep(data_c, data_p, all_data=None, base_map=None)
    init_params(name='')
    model_name = 'default'
    product_gls()

class DecayGroup(chains)
    Bases: DecayGroup, AmpBase
    A Group of Decay Chains with the same final particles.
    add_used_chains(used_chains)
    amp_index(gen=None, base_map=None)
    chains_particle()
    factor_iteration(deep=2)
    generate_phasespace(num=100000)
    get_amp(data)
        calculate the amplitude as complex number
    get_amp2(data)
    get_amp3(data)

```

`get_angle_amp(data)`
`get_base_map(gen=None, base_map=None)`
`get_density_matrix()`
`get_factor()`
`get_factor_angle_amp(data)`
`get_factor_variable()`
`get_id_swap_transpose(key, n)`
`get_m_dep(data)`
 get mass dependent items
`get_res_map()`
`get_swap_factor(key)`
`get_swap_transpose(trans, n)`
`init_params(name="")`
`partial_weight(data, combine=None)`
`partial_weight_interference(data)`
`set_used_chains(used_chains)`
`set_used_res(res, only=False)`
`sum_amp(data, cached=True)`
 calculat the amplitude modular square
`sum_amp_polarization(data)`
 sum amplitude suqare with density `_get_cg_matrix`

$$P = \sum_{m,m',\dots} A_{m,\dots} \rho_{m,m'} A_{m',\dots}^*$$

`sum_with_polarization(amp)`
`temp_used_res(res)`

class `FloatParams`(`x=0, /`)

Bases: `float`

class `HelicityDecay`(*args, `has_barrier_factor=True, l_list=None, barrier_factor_mass=False, has_ql=True, has_bprime=True, aligned=False, allow_cc=True, ls_list=None, barrier_factor_norm=False, params_polar=None, below_threshold=False, force_min_l=False, params_head=None, no_q0=False, helicity_inner_full=False, ls_selector=None, **kwargs`)

Bases: `AmpDecay`

default decay model

The total amplitude is

$$A = H_{\lambda_B, \lambda_C}^{A \rightarrow B+C} D_{\lambda_A, \lambda_B - \lambda_C}^{J_A^*}(\varphi, \theta, 0)$$

The helicity coupling is

$$H_{\lambda_B, \lambda_C}^{A \rightarrow B+C} = \sum_{ls} g_{ls} \sqrt{\frac{2l+1}{2J_A+1}} \langle l0; s\delta | J_A \delta \rangle \langle J_B \lambda_B; J_C - \lambda_C | s\delta \rangle q^l B_l'(q, q_0, d)$$

The fit parameters is g_{ls}

There are some options

- (1). `has_bprime=False` will remove the $B_l'(q, q_0, d)$ part.
- (2). `has_barrier_factor=False` will remove the $q^l B_l'(q, q_0, d)$ part.
- (3). `barrier_factor_norm=True` will replace q^l with $(q/q_0)^l$
- (4). `below_threshold=True` will replace the mass used to calculate q_0 with

$$m_0^{eff} = m^{min} + \frac{m^{max} - m^{min}}{2} \left(1 + \tanh \frac{m_0 - \frac{m^{max} + m^{min}}{2}}{m^{max} - m^{min}} \right)$$

- (5). `l_list=[l1, l2]` and `ls_list=[[l1, s1], [l2, s2]]` options give the list of all possible LS used in the decay.
- (6). `no_q0=True` will set the $q_0 = 1$.

add_algin(*ret, data*)

build_ls2hel_eq()

build_simple_data()

check_valid_jp()

factor_iter_names(*deep=1, extra=[]*)

get_amp(*data, data_p, **kwargs*)

get_angle_amp(*data, data_p, **kwargs*)

get_angle_g_ls()

get_angle_helicity_amp(*data, data_p, **kwargs*)

get_angle_ls_amp(*data, data_p, **kwargs*)

get_barrier_factor(*mass, q, q0, d*)

get_barrier_factor2(*mass, q2, q02, d*)

get_barrier_factor_mass(*mass*)

get_cg_matrix(*out_sym=False*)

The matrix indexed by $[(l, s), (\lambda_b, \lambda_c)]$. The matrix element is

$$\sqrt{\frac{2l+1}{2j_a+1}} \langle j_b, j_c, \lambda_b, -\lambda_c | s, \lambda_b - \lambda_c \rangle \langle l, s, 0, \lambda_b - \lambda_c | j_a, \lambda_b - \lambda_c \rangle$$

This is actually the pre-factor of $g_l s$ in the amplitude formula.

Returns

2-d array of real numbers

get_factor()

get_factor_H(*data, data_p, **kwargs*)

get_factor_angle_amp(*data, data_p, **kwargs*)

get_factor_angle_helicity_amp(*data, data_p, **kwargs*)

get_factor_m_dep(*data, data_p, **kwargs*)

get_factor_variable()

get_g_ls()

get_helicity_amp(*data, data_p, **kwargs*)

get_ls_amp(*data, data_p, **kwargs*)

get_ls_amp_org(*data, data_p, **kwargs*)

get_ls_list()

get possible ls for decay, with l_list filter possible l

get_m_dep(*data, data_p, **kwargs*)

get_params_head()

get_relative_momentum(*data, from_data=False*)

get_relative_momentum2(*data, from_data=False*)

get_total_ls_list()

init_params()

mask_factor_vars()

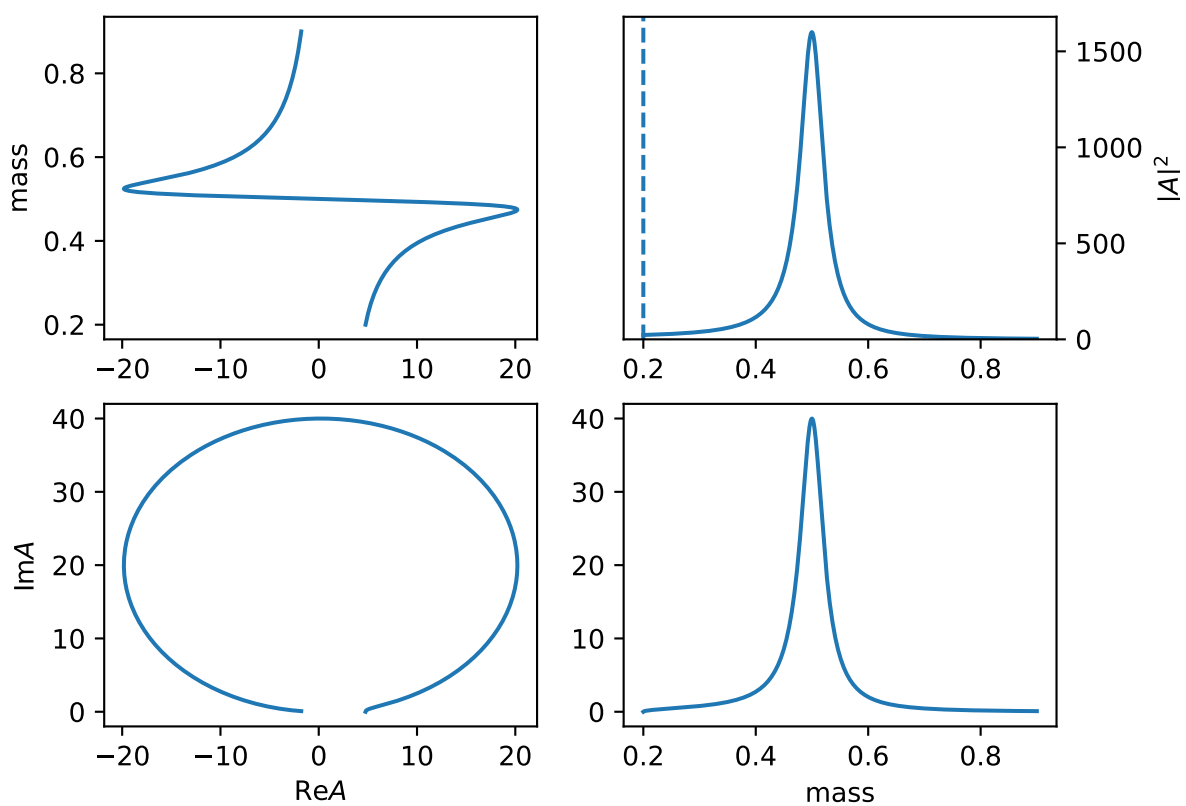
model_name = 'default'

set_ls(*ls*)

class Particle(*args, *running_width=True, bw_l=None, width_norm=False, params_head=None, **kwargs*)

Bases: [BaseParticle](#), [AmpBase](#)

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



```
amp_shape()
get_amp(data, data_c, **kwargs)
get_factor()
get_mass()
get_num_var()
get_subdecay_mass(idx=0)
get_sympy_dom(m, m0, g0, m1=None, m2=None, sheet=0)
get_sympy_var()
get_width()
init_params()
is_fixed_shape()
model_name = 'BWR'
solve_pole(init=None, sheet=0, return_complex=True)

class ParticleX(*args, running_width=True, bw_l=None, width_norm=False, params_head=None, **kwargs)
    Bases: Particle
    simple particle model for mass, (used in expr)


$$R(m) = m$$


get_amp(data, *args, **kwargs)
model_name = 'x'

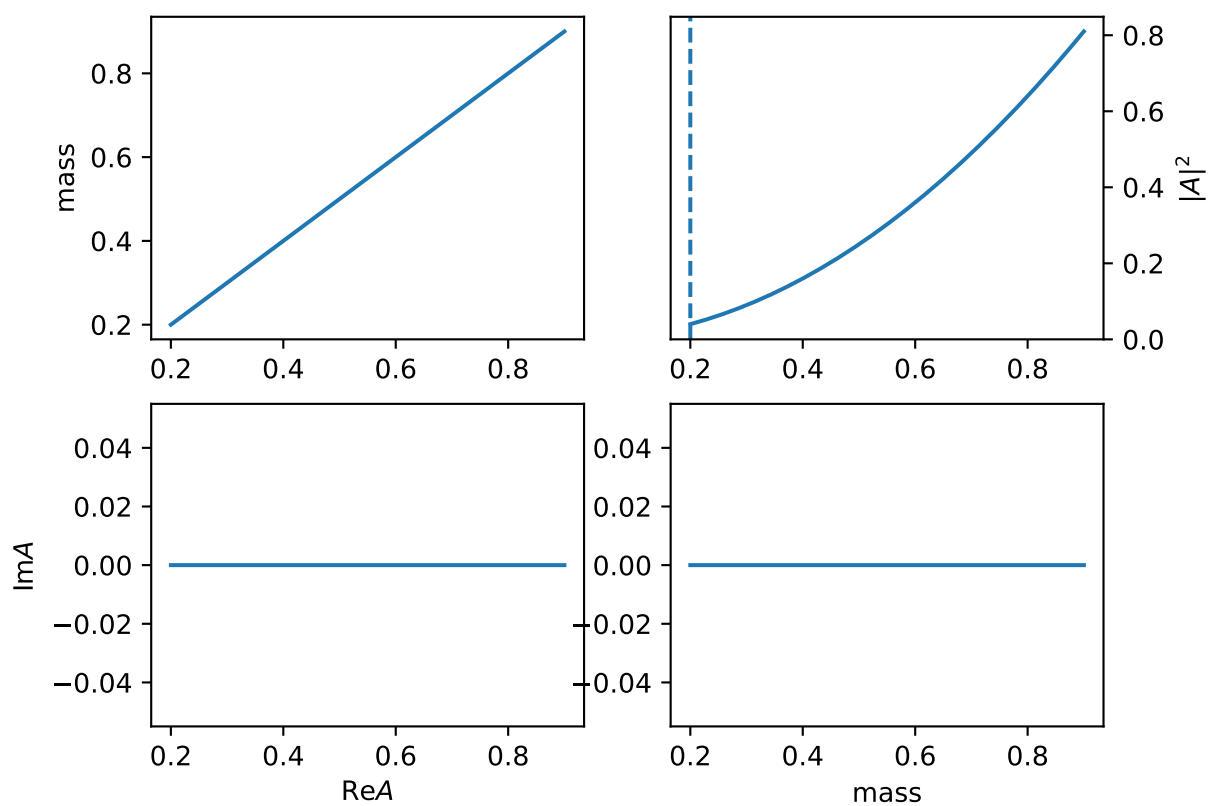
class SimpleResonances(*args, **kwargs)
    Bases: Particle
    get_amp(*args, **kwargs)

data_device(data)

get_decay(core, outs, **kwargs)
    method for getting decay of model
get_decay_chain(decays, **kwargs)
    method for getting decay of model
get_decay_model(model, num_outs=2)

get_name(self, names)

get_particle(*args, model='default', **kwargs)
    method for getting particle of model
get_particle_model(name)
```

get_particle_model_name(*p*)

get_relative_p(*m_0*, *m_1*, *m_2*)

relative momentum for 0 -> 1 + 2

get_relative_p2(*m_0*, *m_1*, *m_2*)

relative momentum for 0 -> 1 + 2

index_generator(*base_map=None*)

load_decfile_particle(*fname*)

ls_selector_qr(*decay*, *ls_list*)

regist_decay(*name=None*, *num_outs=2*, *f=None*)

register a decay model

Params name

model name used in configuration

Params f

Model class

regist_particle(*name=None*, *f=None*)

register a particle model

Params name

model name used in configuration

Params f

Model class

register_decay(*name=None*, *num_outs=2*, *f=None*)

register a decay model

Params name

model name used in configuration

Params f

Model class

register_decay_chain(*name=None*, *f=None*)

register a decay model

Params name

model name used in configuration

Params f

Model class

register_particle(*name=None*, *f=None*)

register a particle model

Params name

model name used in configuration

Params f

Model class

rename_data_dict(*data*, *idx_map*)

simple_cache_fun(*f*)

simple_deepcopy(*dic*)

simple_resonance(*name, fun=None, params=None*)

convert simple fun *f*(*m*) into a resonances model

Params name

model name used in configuration

Params fun

Model function

Params params

arguments name list for parameters

trans_model(*model*)

value_and_grad(*f, var*)

variable_scope(*vm=None*)

variabel name scope

cov_ten

class CovTenDecayChain(*args, *is_cp=False, aligned=True, **kwargs*)

Bases: [DecayChain](#)

build_coupling_einsum(*a, b, c, na, nb, nc, idx_map*)

build_decay_einsum(*ls, idx_map=None*)

build_einsum()

build_l_einsum(*decay, l, s, idx_map*)

build_s_einsum(*decay, l, s, idx_map*)

build_wave_function(*particle, pi*)

get_amp(*data_c, data_p, all_data=None, base_map=None, idx_map=None*)

get_finals_amp(*data_p*)

get_m_dep_list(*data_c, data_p, all_data=None*)

init_params(*name=""*)

model_name = 'cov_ten'

class CovTenDecayNew(*args, **kwargs)

Bases: [HelicityDecay](#)

Decay Class for covariant tensor formula

get_amp(*data, data_p, **kwargs*)

get_angle_amp(*data, data_p, **kwargs*)

```
class CovTenDecaySimple(*args, **kwargs)
    Bases: CovTenDecayNew
    Decay Class for covariant tensor formula
    get_all_amp(data, data_p, **kwargs)
    get_amp(data, data_p, **kwargs)
    init_params()
    model_name = 'cov_ten_simple'

class EinSum(name, idx, inputs=None, replace_axis=[])
    Bases: object
    call(inputs, cached=None)
    insert_extra_axis(name, indexs)
    set_inputs(name, value)

class EinSumCall(name, idx, function)
    Bases: EinSum
    call(inputs, cached=None)

class EvalBoost(boost, sign=None)
    Bases: object
    call(inputs, cached=None)

class EvalP(core, l)
    Bases: object
     $P^{\{u\}}_{\{v\}}$ 

class EvalT(decay, l)
    Bases: object
     $t^{\{u\}}$ 

class EvalT2(decay, l)
    Bases: object
     $t^{\{u\}}$ 

class IndexMap
    Bases: object
    get(name)

create_epsilon()
     $\epsilon^{\{a\}}_{\{bcd\}}$ 

dot(p1, p2)

mass2(t)

wave_function(J, p)
```

flatte

class ParticleFlatte2(*args, im_sign=-1, l_list=None, has_bprime=True, no_m0=False, no_q0=False, cut_phsp=False, **kwargs)

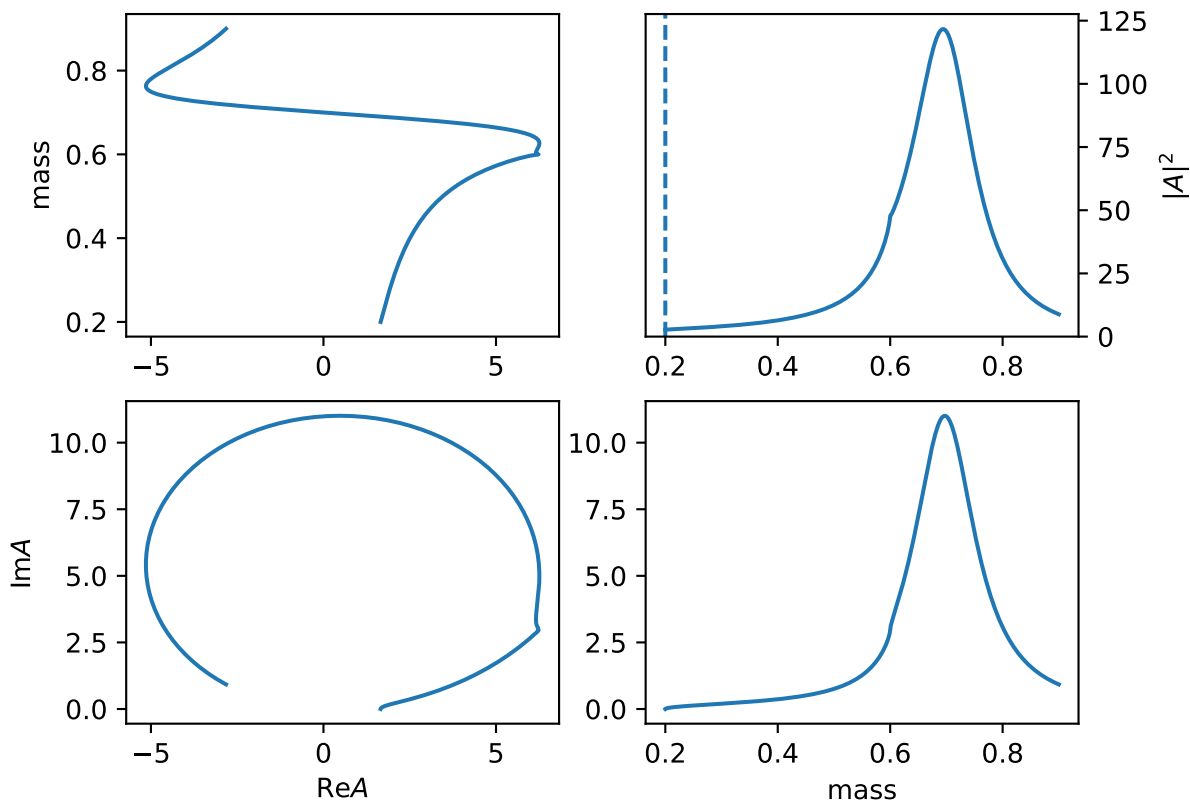
Bases: [ParticleFlatteGen](#)

General Flatte like formula.

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0[\sum_i g_i^2 \frac{q_i}{m} \times \frac{m_0}{|q_{i0}|} \times \frac{|q_i|^{2l_i}}{|q_{i0}|^{2l_i}} B_{l_i}'(|q_i|, |q_{i0}|, d)]}$$

$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ \frac{i\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$

It has the same options as [FlatteGen](#).



get_coeff()

model_name = 'Flatte2'

class ParticleFlatteGen(*args, im_sign=-1, l_list=None, has_bprime=True, no_m0=False, no_q0=False, cut_phsp=False, **kwargs)

Bases: [ParticleFlatte](#)

More General Flatte like formula

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0[\sum_i g_i \frac{q_i}{m} \times \frac{m_0}{|q_{i0}|} \times \frac{|q_i|^{2l_i}}{|q_{i0}|^{2l_i}} B_{l_i}'^2(|q_i|, |q_{i0}|, d)]}$$

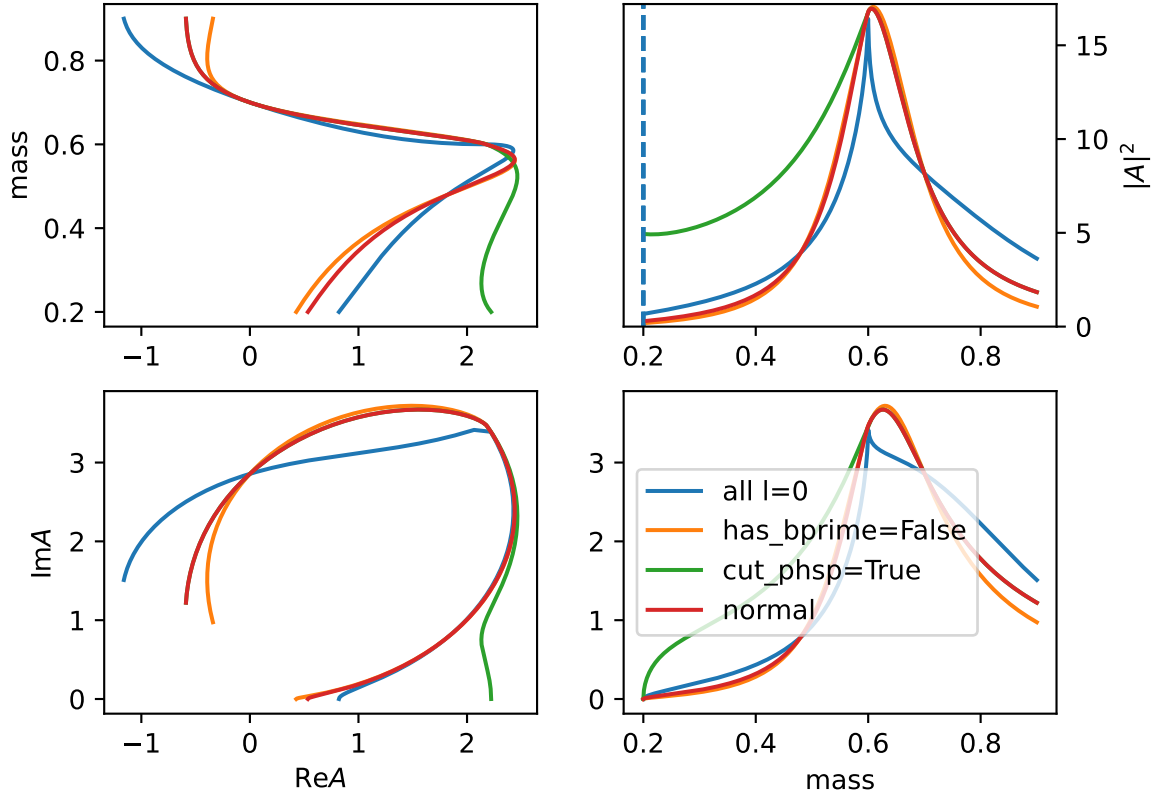
$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$

Required input arguments `mass_list`: `[[m11, m12], [m21, m22]]` for $m_{i,1}, m_{i,2}$. And addition arguments `l_list`: `[l1, l2]` for l_i

`has_bprime=False` to remove $B_{l_i}'^2(|q_i|, |q_{i0}|, d)$.

`cut_phsp=True` to set $q_i = 0$ when $(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0$

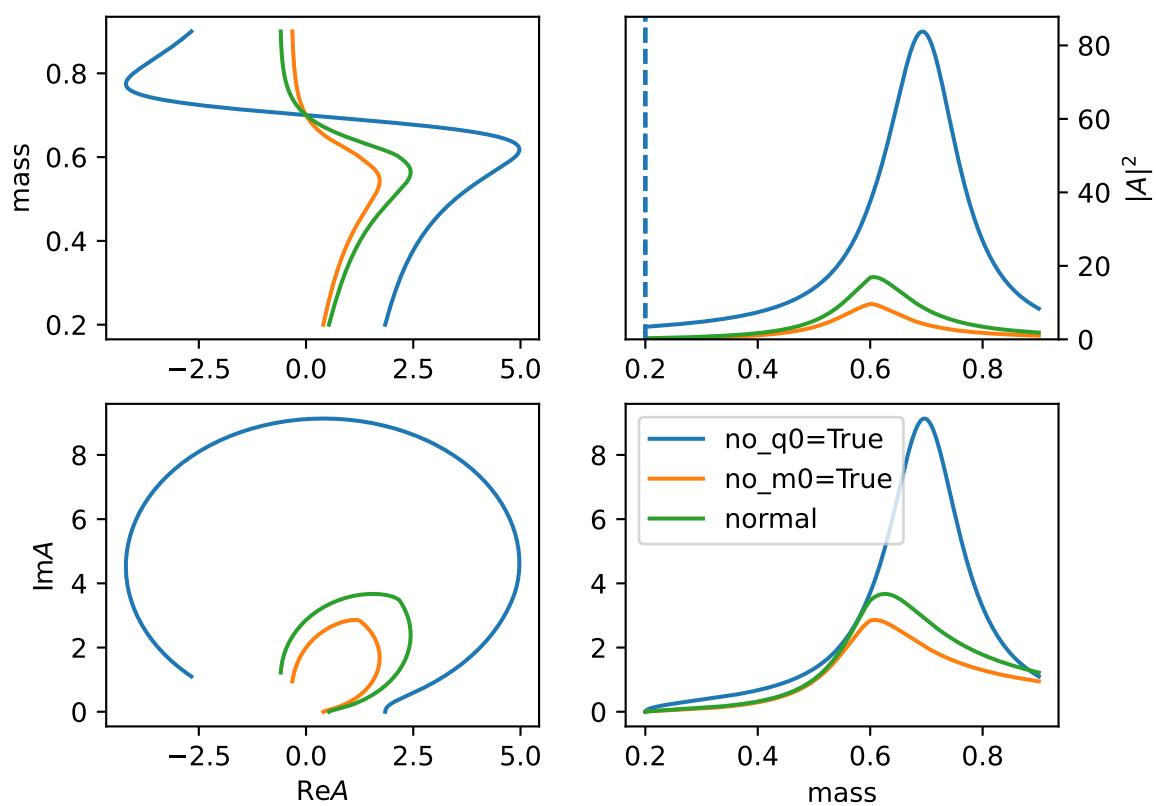
The plot use parameters $m_0 = 0.7, m_{0,1} = m_{0,2} = 0.1, m_{1,1} = m_{1,2} = 0.3, g_0 = 0.3, g_1 = 0.2, l_0 = 0, l_1 = 1$.



`no_m0=True` to set $im_0 \Rightarrow i$ in the width part.

`no_q0=True` to remove $\frac{m_0}{|q_{i0}|}$ and set others $q_{i0} = 1$.

`get_amp(*args, **kwargs)`



```

get_coeff()

get_num_var()

get_sympy_dom(m, m0, *gi, sheet=0)

model_name = 'FlatteGen'

```

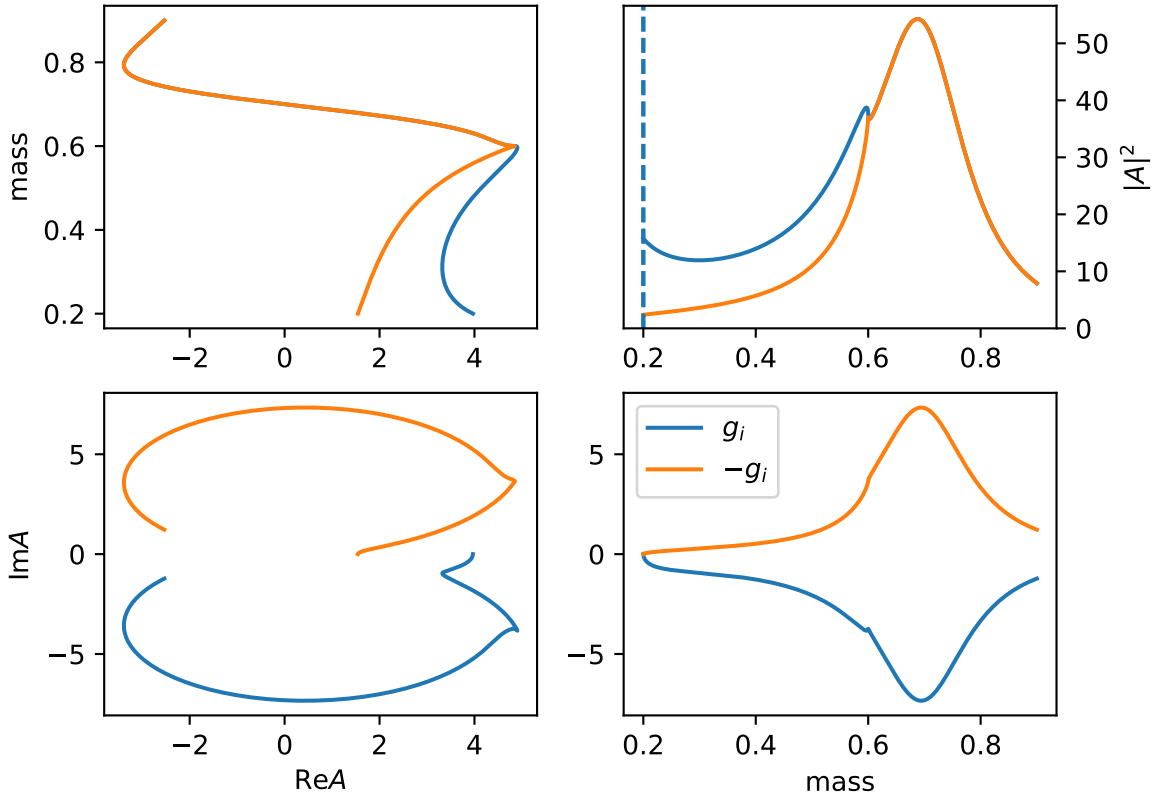
```
class ParticleFlatte(*args, mass_list=None, im_sign=1, **kwargs)
```

Bases: [Particle](#)

Flatte like formula

$$R(m) = \frac{1}{m_0^2 - m^2 + im_0(\sum_i g_i \frac{q_i}{m})}$$

$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$



Required input arguments `mass_list`: `[[m11, m12], [m21, m22]]` for $m_{i,1}, m_{i,2}$.

```
get_amp(*args, **kwargs)
```

```
get_num_var()
```



```
get_sympy_dom(m, m0, *gi, sheet=0)
```

```
get_sympy_var()
```

```
get_width(m=None)
```

```
init_params()
```

```
model_name = 'Flatte'
```

```
class ParticleFlatteC(*args, im_sign=-1, **kwargs)
```

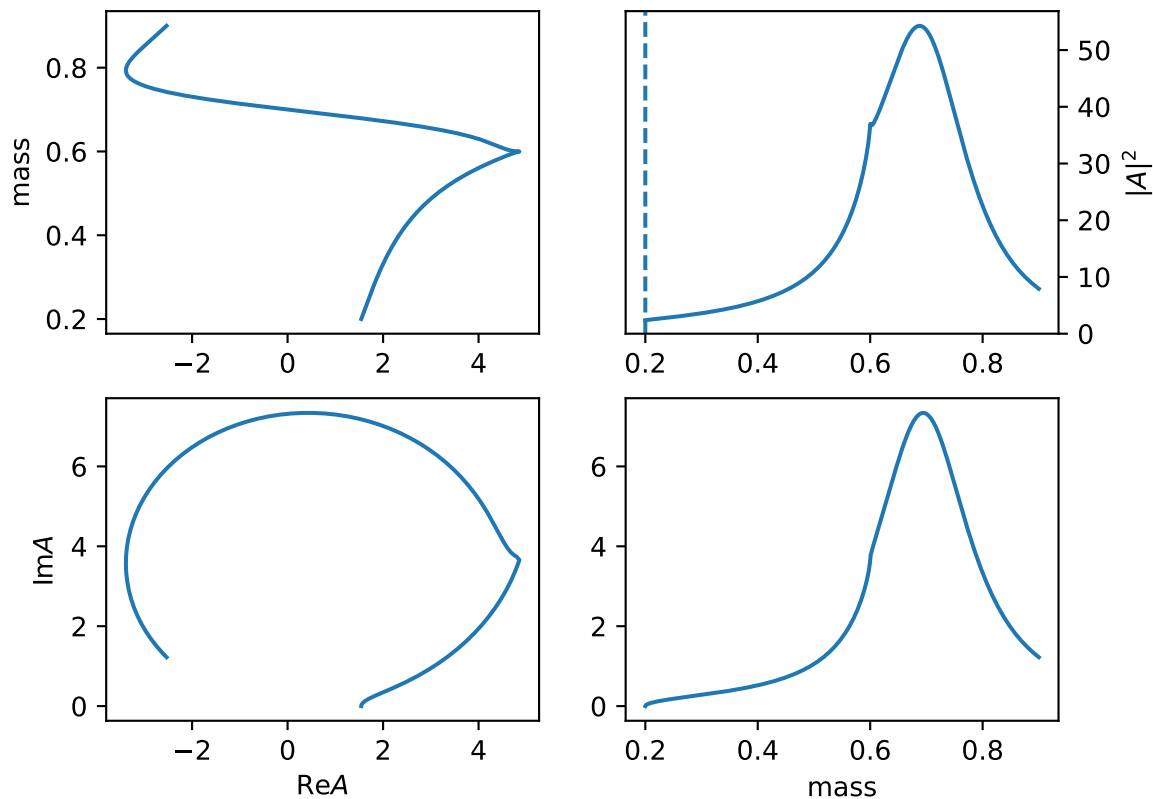
Bases: [ParticleFlatte](#)

Flatte like formula

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0(\sum_i g_i \frac{q_i}{m})}$$

$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$

Required input arguments `mass_list`: `[[m11, m12], [m21, m22]]` for $m_{i,1}, m_{i,2}$.



```
model_name = 'FlatteC'
```

```
cal_monentum(m, ma, mb)
```

```
cal_monentum_sympy(m, ma, mb)
```

interpolation

```
class HistParticle(*args, **kwargs)
```

Bases: *InterpolationParticle*

```
n_points()
```

```
class Interp(*args, **kwargs)
```

Bases: *InterpolationParticle*

linear interpolation for complex number

```
interp(m)
```

```
model_name = 'interp_c'
```

```
class Interp1D3(*args, **kwargs)
```

Bases: *InterpolationParticle*

Piecewise third order interpolation

```
interp(m)
```

```
model_name = 'interp1d3'
```

```
class Interp1DLang(*args, **kwargs)
```

Bases: *InterpolationParticle*

Lagrange interpolation

```
interp(m)
```

```
model_name = 'interp_lagrange'
```

```
class Interp1DSpline(*args, **kwargs)
```

Bases: *InterpolationParticle*

Spline interpolation function for model independent resonance

```
init_params()
```

```
interp(m)
```

```
model_name = 'spline_c'
```

```
class Interp1DSplineIdx(*args, **kwargs)
```

Bases: *InterpolationParticle*

Spline function in index way.

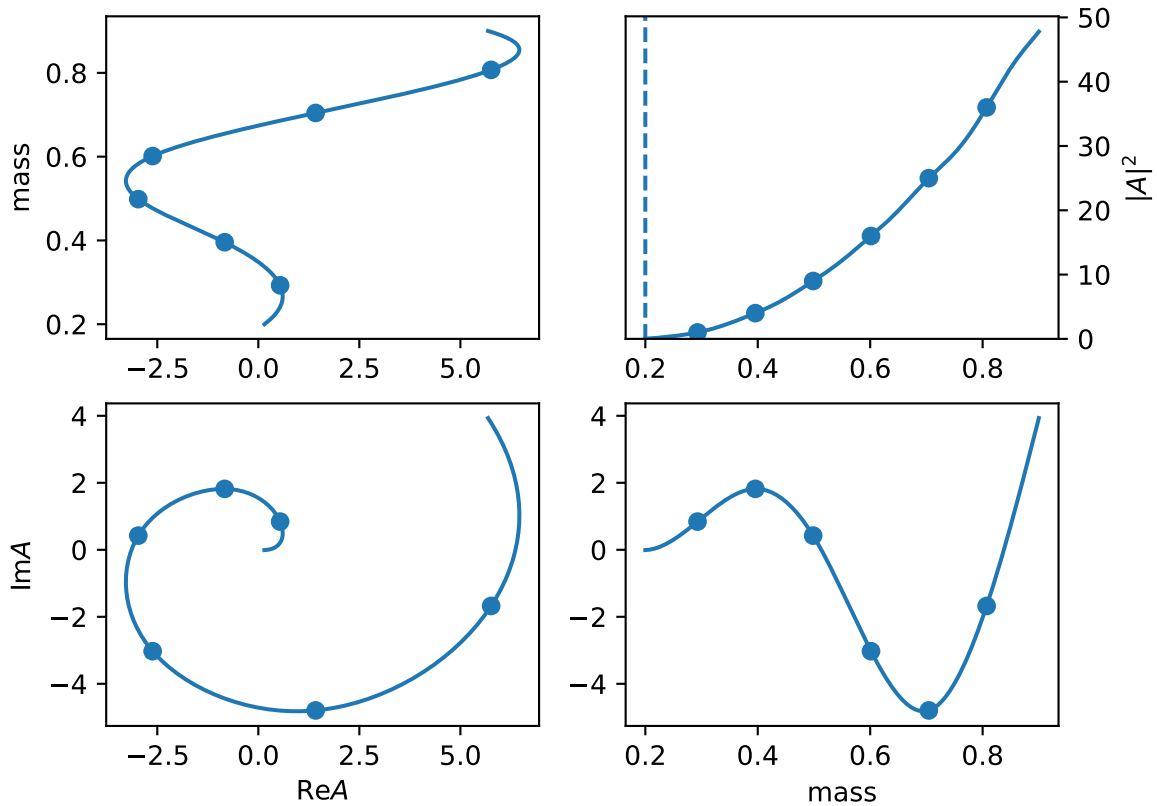
use

```
min_m: 0.19
max_m: 0.91
interp_N: 8
with_bound: True
```

for mass range [0.19, 0.91] and 8 interpolation points

The first and last are fixed to zero unless set `with_bound: True`.

This is an example of $k \exp(ik)$ for point k .



```
init_params()
```

```
interp(m)
```

```
model_name = 'spline_c_idx'
```

```
class InterpHist(*args, **kwargs)
```

Bases: [InterpolationParticle](#)

Interpolation for each bins as constant

```
interp(m)
```

```
model_name = 'interp_hist'
```

```
class InterpHistIdx(*args, **kwargs)
```

Bases: [HistParticle](#)

Interpolation for each bins as constant

use

```

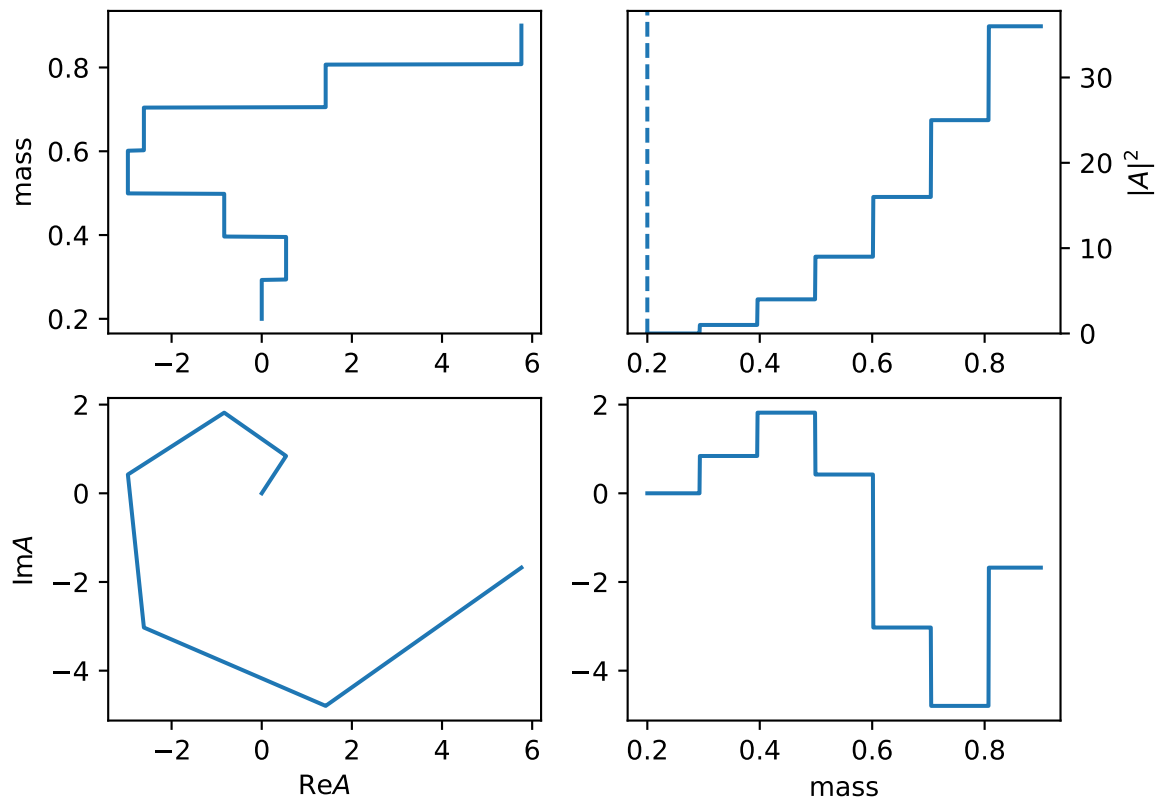
min_m: 0.19
max_m: 0.91
interp_N: 8
with_bound: True

```

for mass range [0.19, 0.91] and 7 bins

The first and last are fixed to zero unless set with_bound: True.

This is an example of $k \exp(ik)$ for point k.



```
interp(m)
```

```
model_name = 'hist_idx'
```

```
class InterpL3(*args, **kwargs)
```

```
Bases: InterpolationParticle
```

```
interp(m)
```

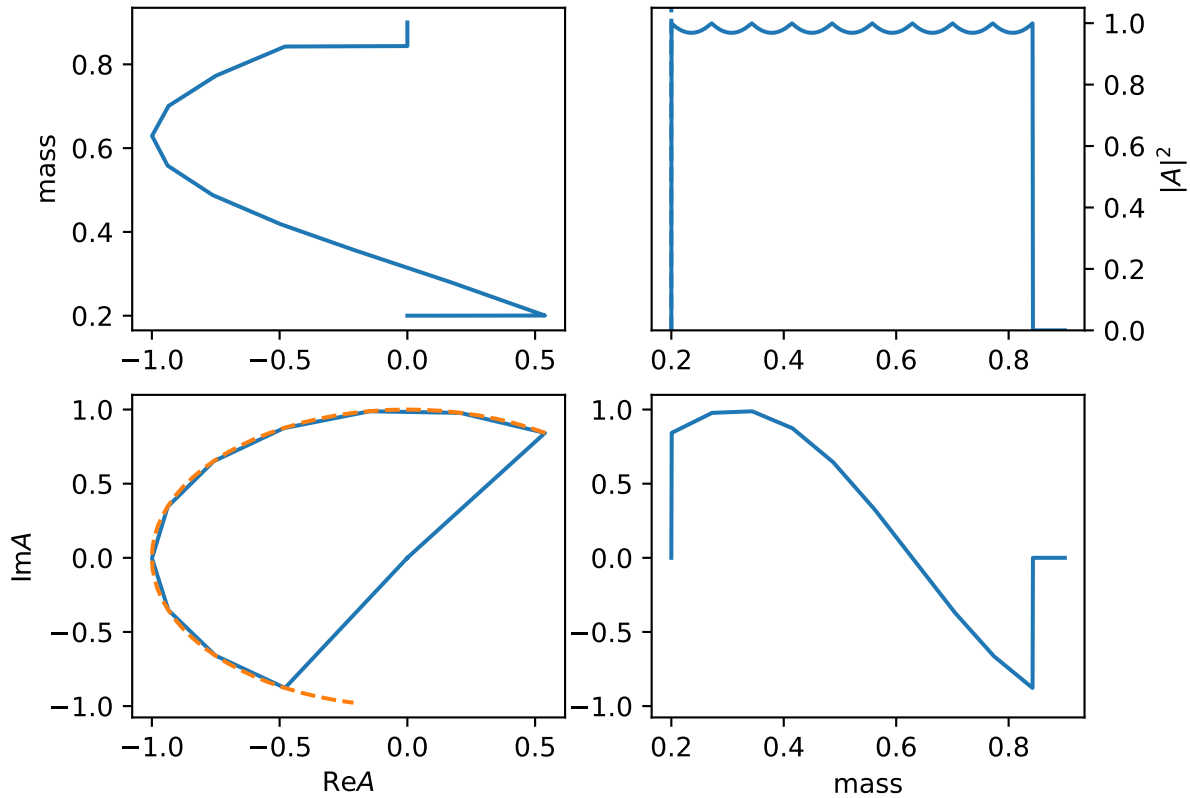
```
model_name = 'interp_l3'
```

```
class InterpLinearNpy(*args, **kwargs)
```

```
Bases: InterpolationParticle
```

Linear interpolation model from a npy file with array of [mi, re(ai), im(ai)]. Required file: path_of_file.npy, for the path of npy file.

The example is $\exp(5 \text{ I } m)$.



```
get_point_values()
```

```
init_params()
```

```
interp(m)
```

```
model_name = 'linear_npy'
```

```
class InterLinearTxt(*args, **kwargs)
```

Bases: [InterLinearNpy](#)

Linear interpolation model from a txt file with array of $[m_i, \text{re}(a_i), \text{im}(a_i)]$. Required file: `path_of_file.txt`, for the path of txt file.

The example is $\exp(5 \text{ I } m)$.

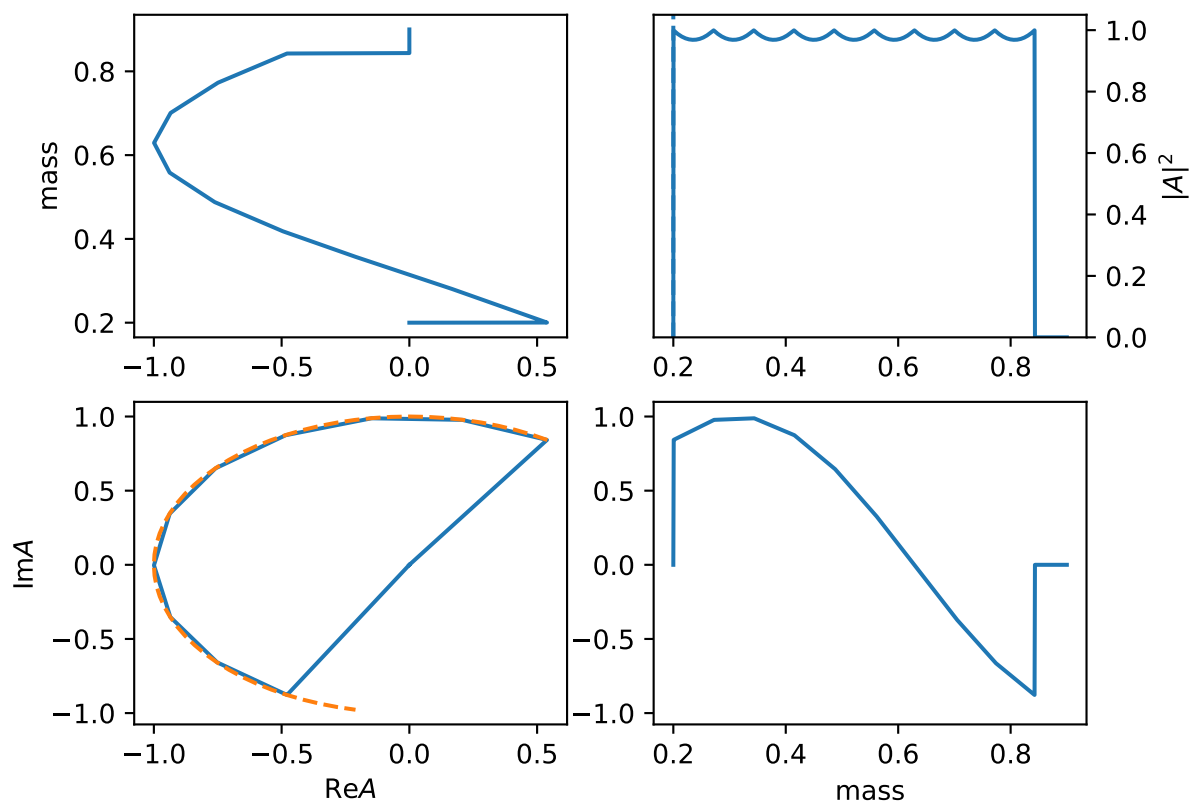
```
model_name = 'linear_txt'
```

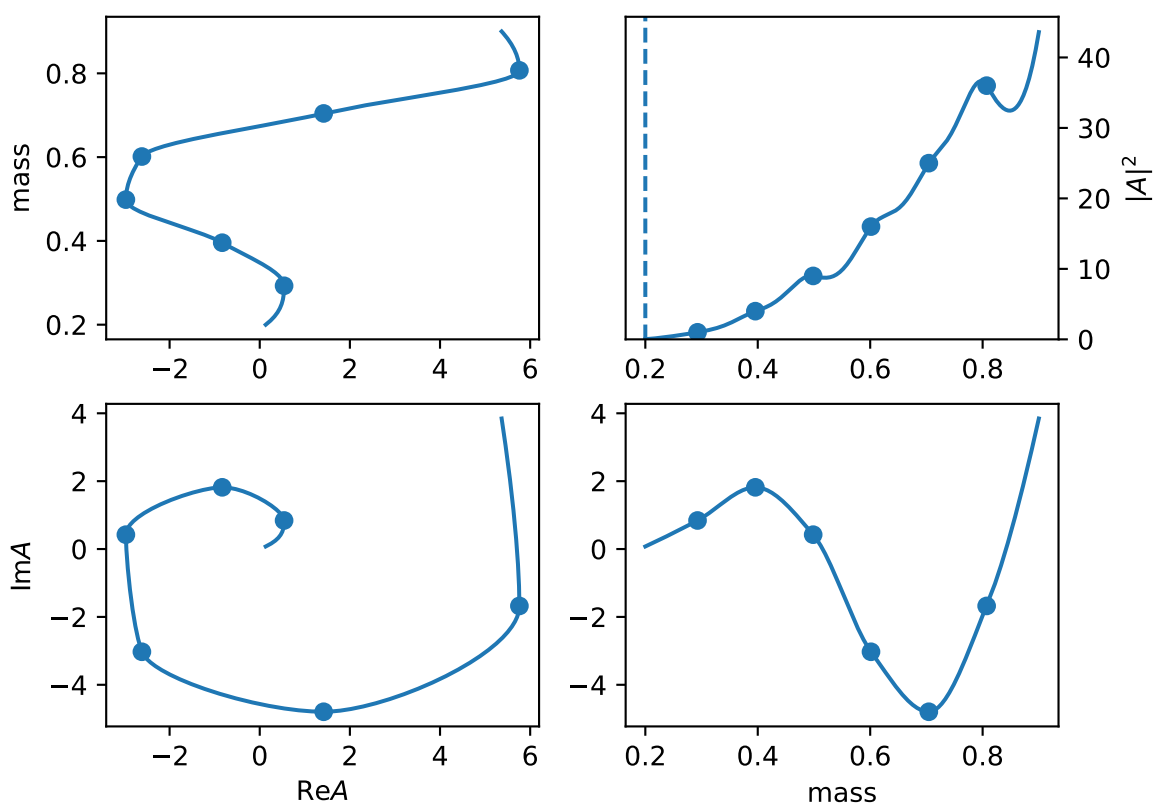
```
class InterpSPPCHIP(*args, **kwargs)
```

Bases: [InterpolationParticle](#)

Shape-Preserving Piecewise Cubic Hermite Interpolation Polynomial. It is monotonic in each interval.

```
init_params()
```





```
interp(m)

model_name = 'sppchip'

class InterpolationParticle(*args, **kwargs)
    Bases: Particle
    get_amp(data, *args, **kwargs)
    get_bin_index(m)
    get_point_values()
    init_params()
    interp(mass)
    n_points()

create_sppchip_matrix(points)
    matrix to solve  $f(x_i), f(x_{i+1}), f'(x_i), f'(x_{i+1})$ 

do_spline_hmatrix(h_matrix, y, m, idx)

get_matrix_interp1d3(x, xi)

get_matrix_interp1d3_v2(x, xi)

interp1d3(x, xi, yi)

spline_matrix(x, xi, yi, bc_type='not-a-knot')
    calculate spline interpolation

spline_x_matrix(x, xi)
    build matrix of x for spline interpolation

spline_xi_matrix(xi, bc_type='not-a-knot')
    build matrix of xi for spline interpolation solve equation
```

$$S'_i(x_i) = S'_{i-1}(x_i)$$

and two bound condition. $S'_0(x_0) = S'_{n-1}(x_n) = 0$

```
sppchip(m, xi, y, idx=None, matrix=None)
```

Shape-Preserving Piecewise Cubic Hermite Interpolation Polynomial. It is monotonic in each interval.

```
>>> from scipy.interpolate import pchip_interpolate
>>> x_observed = np.linspace(0.0, 10.0, 11)
>>> y_observed = np.sin(x_observed)
>>> x = np.linspace(min(x_observed), max(x_observed)-1e-12, num=100)
>>> y = pchip_interpolate(x_observed, y_observed, x)
>>> assert np.allclose(y, sppchip(x, x_observed, y_observed).numpy())
```

```
sppchip_coeffs(xi, y, matrix=None, eps=1e-12)
```


kmatrix_simple

class KmatrixSimple(*args, **kwargs)

Bases: *KmatrixSplitLSParticle*

simple Kmatrix formula.

K-matrix

$$K_{i,j} = \sum_a \frac{g_{i,a} g_{j,a}}{m_a^2 - m^2 + i\epsilon}$$

P-vector

$$P_i = \sum_a \frac{\beta_a g_{i,a}}{m_a^2 - m^2 + i\epsilon} + f_{bkg,i}$$

total amplitude

$$R(m) = n(1 - Ki\rho n^2)^{-1}P$$

barrier factor

$$n_{ii} = q_i^l B_l'(q_i, 1/d, d)$$

phase space factor

$$\rho_{ii} = q_i/m$$

q_i is 0 when below threshold

build_barrier_factor(s)

build_k_matrix(s)

build_p_vector(s)

get_ls_amp(m)

init_params()

model_name = 'KmatrixSimple'

phsp_fractor(m, m1, m2)

barrier_factor(m, m1, m2, l, d=3.0)

get_relative_p(m, m1, m2)

preprocess

class BasePreProcessor(decay_struct, root_config=None, model='default', **kwargs)

Bases: *HeavyCall*

class CachedAmpPreProcessor(*args, **kwargs)

Bases: *BasePreProcessor*

class CachedAnglePreProcessor(*args, **kwargs)

Bases: *BasePreProcessor*

build_cached(*x*)

class **CachedShapePreProcessor**(*args, **kwargs)

Bases: [CachedAmpPreProcessor](#)

build_cached(*x*)

create_preprocessor(*decay_group*, **kwargs)

list_to_tuple(*data*)

register_preprocessor(*name=None*, *f=None*)

register a data mode

Params name

mode name used in configuration

Params f

Data Mode class

split_ls

class **ParticleBWRLS**(*args, **kwargs)

Bases: [ParticleLS](#)

Breit Wigner with split ls running width

$$R_i(m) = \frac{g_i}{m_0^2 - m^2 - im_0\Gamma_0 \frac{\rho}{\rho_0} (\sum_i g_i^2)}$$

, $\rho = 2q/m$, the partial width factor is

$$g_i = \gamma_i \frac{q^l}{q_0^l} B'_{l_i}(q, q_0, d)$$

and keep normalize as

$$\sum_i \gamma_i^2 = 1.$$

The normalize is done by $(\cos \theta_0, \sin \theta_0 \cos \theta_1, \dots, \prod_i \sin \theta_i)$

factor_gamma(*ls*)

get_barrier_factor(*ls*, *q2*, *q02*, *d*)

get_ls_amp(*m*, *ls*, *q2*, *q02*, *d=3.0*)

get_ls_amp_frac(*m*, *ls*, *q2*, *q02*, *d=3.0*)

get_num_var()

get_sympy_dom(*m*, *m0*, *g0*, *thetas*, *m1=None*, *m2=None*, *sheet=0*)

get_sympy_var()

init_params()

model_name = 'BWR_LS'

```
class ParticleBWRLS2(*args, **kwargs)
```

Bases: [ParticleLS](#)

Breit Wigner with split ls running width, each one use their own l,

$$R_i(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0 \frac{\rho}{\rho_0}(g_i^2)}$$

, $\rho = 2q/m$, the partial width factor is

$$g_i = \gamma_i \frac{q^l}{q_0^l} B'_{l_i}(q, q_0, d)$$

```
    get_ls_amp(m, ls, q2, q02, d=3.0)
```

```
    model_name = 'BWR_LS2'
```

```
class ParticleDecayLS(*args, **kwargs)
```

Bases: [HelicityDecay](#)

```
    get_barrier_factor2(mass, q2, q02, d)
```

```
    init_params()
```

```
    model_name = 'LS-decay'
```

```
class ParticleLS(*args, **kwargs)
```

Bases: [Particle](#)

```
    get_amp(*args, **kwargs)
```

```
    get_ls_amp(m, ls, q2, q02, d=3)
```

```
    is_fixed_shape()
```

```
class ParticleMultiBW(*args, **kwargs)
```

Bases: [ParticleMultiBWR](#)

Combine Multi BW into one particle

```
    dom_fun(m, m0, g0, q2, q02, l, d)
```

```
    model_name = 'MultiBW'
```

```
class ParticleMultiBWR(*args, **kwargs)
```

Bases: [ParticleLS](#)

Combine Multi BWR into one particle

```
    dom_fun(m, m0, g0, q2, q02, l, d)
```

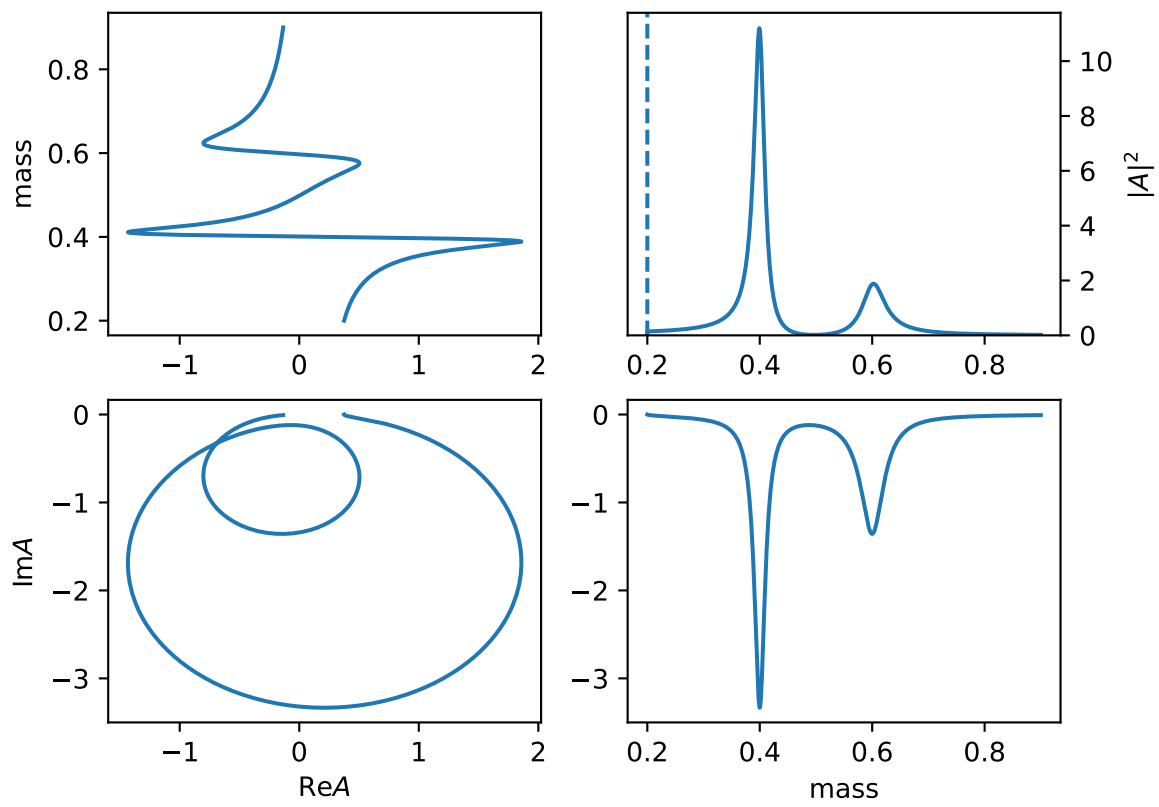
```
    get_barrier_factor(ls, q2, q02, d)
```

```
    get_ls_amp(m, ls, q2, q02, d=3.0)
```

```
    init_params()
```

```
    mass()
```

```
    model_name = 'MultiBWR'
```



10.1.2 app

Submodules and Subpackages

fit

fit(*config*='config.yml', *init_params*='init_params.json', *method*='BFGS')
 simple fit script

json_print(*dic*)
 print parameters as json

10.1.3 config_loader

Submodules and Subpackages

base_config

class BaseConfig(*file_name*, *share_dict*=None)
 Bases: `object`

load_config(*file_name*, *share_dict*=None)

config_loader

class ConfigLoader(*file_name*, *vm*=None, *share_dict*=None)
 Bases: `BaseConfig`

class for loading config.yml

add_constraints(*amp*)

add_decay_constraints(*amp*, *dic*=None)

add_fix_var_constraints(*amp*, *dic*=None)

add_free_var_constraints(*amp*, *dic*=None)

add_from_trans_constraints(*amp*, *dic*=None)

add_gauss_constr_constraints(*amp*, *dic*=None)

add_particle_constraints(*amp*, *dic*=None)

add_pre_trans_constraints(*amp*, *dic*=None)

add_var_equal_constraints(*amp*, *dic*=None)

add_var_range_constraints(*amp*, *dic*=None)

attach_fix_params_error(*params: dict, V_b=None*) → ndarray

The minimal condition

$$-\frac{\partial \ln L(a, b)}{\partial a} = 0,$$

can be treated as a implicit function $a(b)$. The gradients is

$$\frac{\partial a}{\partial b} = -\left(\frac{\partial^2 \ln L(a, b)}{\partial a \partial a}\right)^{-1} \frac{\partial \ln L(a, b)}{\partial a \partial b}.$$

The uncertainties from b with error matrix V_b can propagate to a as

$$V_a = \frac{\partial a}{\partial b} V_b \frac{\partial a}{\partial b}$$

This matrix will be added to the config.inv_he.

batch_sum_var(*args, **kwargs)

cal_bins_numbers(*adapter, data, phsp, read_data, bg=None, bg_weight=None*)

cal_chi2(*read_data=None, bins=[[2, 2], [2, 2], [2, 2]], mass=['R_BD', 'R_CD']*)

cal_fitfractions(*params={}, mcdata=None, res=None, exclude_res=[], batch=25000, method='old'*)

cal_signal_yields(*params={}, mcdata=None, batch=25000*)

check_valid_jp(*decay_group*)

eval_amplitude(*p, *extra=None*)

fit(*data=None, phsp=None, bg=None, inmc=None, batch=65000, method='BFGS', check_grad=False, improve=False, reweight=False, maxiter=None, jac=True, print_init_nll=True, callback=None, grad_scale=1.0, gtol=0.001*)

fitNtimes(*N, *args, **kwargs*)

free_for_extended(*amp*)

generate_SDP(*node, N=1000, include_charge=False, legacy=True*)

generate_SDP_p(*node, N=1000, legacy=False*)

generate_phsp(*N=1000, include_charge=False, cal_max=False*)

generate_phsp_p(*N=1000, cal_max=False*)

generate_toy(*N=1000, force=True, gen=None, gen_p=None, importance_f=None, max_N=100000, include_charge=False, cal_phsp_max=False*)

A more accurate method for generating toy data.

Parameters

- **N** – number of events.
- **force** – if remove extra data generated.
- **gen** – optional function for generate phase space, the return value is same as config.get_data.
- **gen_p** – optional function for generate phase space, the return value is dict as {B: pb, C: pc, D: pd}.
- **max_N** – max number of events for every try.

```

generate_toy2(*args, **kwargs)

generate_toy_o(N=1000, force=True, max_N=100000)

generate_toy_p(N=1000, force=True, gen_p=None, importance_f=None, max_N=100000,
                include_charge=False, cal_phsp_max=False)
    generate toy data momentum.

get_SDP_generator(node, include_charge=False, legacy=True)

get_SDP_p_generator(node, legacy=True)

get_all_data()

get_all_frame()

get_all_plotdatas(data=None, phsp=None, bg=None, res=None, use_weighted=False)

get_amplitude(vm=None, name="")

get_chain(idx)

get_chain_property(idx, display=True)
    Get chain name and curve style in plot

get_dalitz(a, b)

get_dalitz_boundary(a, b, N=1000)

get_dat_order(standard=False)

get_data(idx)

get_data_file(idx)

get_data_index(sub, name)

get_data_rec(name)

get_decay(full=True)

get_fcn(all_data=None, batch=65000, vm=None, name="")

get_ndf()

get_params(trainable_only=False)

get_params_error(params=None, data=None, phsp=None, bg=None, inmc=None, batch=10000,
                  using_cached=False, method=None, force_pos=True, correct_params=None)
    calculate parameters error

get_particle_function(name, d_norm=False)

get_phsp_generator(include_charge=False, nodes=[])

get_phsp_noeff()

get_phsp_p_generator(nodes=[])

get_phsp_plot(tail="")

```

```
get_plotter(legend_file=None, res=None, datasets=None, use_weighted=False)

likelihood_profile(var, var_min, var_max, N=100)

load_cached_data(file_name=None)

static load_config(file_name, share_dict={})

mask_params(params)

params_trans()

plot_adaptive_2dpull(var1, var2, binning=[[2, 2], [2, 2], [2, 2]], ax=<module 'matplotlib.pyplot' from
    '/home/docs/checkouts/readthedocs.org/user_builds/tf-
    pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, where={},
    cut_zero=True, plot_scatter=True, scatter_style={'c': 'black', 's': 1}, **kwargs)

plot_partial_wave(params=None, data=None, phsp=None, bg=None, prefix='figure/', res=None,
    save_root=False, chains_id_method=None, phsp_rec=None, cut_function=<function
    <lambda>>, plot_function=None, **kwargs)
```

plot partial wave plots

Parameters

- **self** – ConfigLoader object
- **params** – params, dict or FitResults
- **data** – data sample, a list of CalAngleData
- **phsp** – phase space sample, a list of CalAngleData (the same size as data)
- **bg** – background sample, a list of CalAngleData (the same size as data)
- **prefix** – figure saving folder and naming prefix
- **res** – combination of resonances in partial wave, list of (list of (string for resonance name or int for decay chain index))
- **save_root** – if save weights in a root file, bool
- **chains_id_method** – method of how legend label display, string
- **bin_scale** – more binning in partial waves for a smooth histogram. int
- **batch** – batching in calculating weights, int
- **smooth** – if plot smooth binned kde shape or histogram, bool
- **single_legend** – if save all legend in a file “legend.pdf”, bool
- **plot_pull** – if plot the pull distribution, bool
- **format** – save figure with image format, string (such as “.png”, “.jpeg”)
- **linestyle_file** – legend linestyle configuration file name (YAML format), string (such as “legend.yml”)

```
plot_partial_wave_interf(res1, res2, **kwargs)
```

```
register_extra_constrains(name, f=None)
```

add extra_constrains

```
classmethod register_function(name=None)
```



```

    reinit_params()

    static reweight_init_value(amp, phsp, ns=None)
        reset decay chain total and make the integration to be ns

    save_cached_data(data, file_name=None)

    save_params(file_name)

    save_tensorflow_model(dir_name)

    set_params(params, neglect_params=None)

class PlotParams(plot_config, decay_struct)
    Bases: dict
    get_angle_vars(is_align=False)
    get_data_index(sub, name)
    get_mass_vars()
    get_params(params=None)

set_prefix_constrains(vm, base, params_dic, self)

validate_file_name(s)

```

data

```

class MultiData(*args, **kwargs)
    Bases: SimpleData
    get_data(idx) → list
    get_n_data()
    get_phsp_noeff()
    process_scale(idx, data)
    set_lazy_call(data, idx)

class SimpleData(dic, decay_struct, config=None)
    Bases: object
    cal_angle(p4, **kwargs)
    get_all_data()
    get_dat_order(standard=False)
    get_data(idx) → dict
    get_data_file(idx)
    get_data_index(sub, name)
    get_n_data()

```

```
get_phsp_noeff()
get_phsp_plot()
get_weight_sign(idx)
load_cached_data(file_name=None)
load_data(files, weight_sign=1, weight_smear=None, **kwargs) → dict
load_extra_var(n_data, **kwargs)
load_p4(fnames)
load_weight_file(weight_files)
process_scale(idx, data)
save_cached_data(data, file_name=None)
savetxt(file_name, data)
set_lazy_call(data, idx)

load_data_mode(dic, decay_struct, default_mode='multi', config=None)

register_data_mode(name=None, f=None)
    register a data mode

    Params name
        mode name used in configuration

    Params f
        Data Mode class
```

data_root_lhcb

```
class RootData(*args, **kwargs)
    Bases: MultiData

    create_data(p4, **kwargs)

    get_data(idx)

    get_p4(idx)

    get_weight(idx)

    load_var(idx, tail)

build_matrix(order, matrix)

custom_cond(x, dic, key=None)

cut_data(data)

touch_var(name, data, var, size, default=1)
```

decay_config

```

class DecayConfig(dic, share_dict={})
    Bases: BaseConfig
    decay_chain_cut(decays)
    decay_chain_cut_list = {}
    decay_cut(decays)
    decay_cut_list = {'ls_cut': <function decay_cut_ls>, 'mass_cut': <function
    decay_cut_mass>}
    static decay_item(decay_dict)
    disable_allow_cc(decay_group)
    get_decay(full=True)
    get_decay_struct(decay, particle_map=None, particle_params=None, top=None, finals=None,
    chain_params={}, process_cut=True)
        get decay structure for decay dict
    static load_config(file_name, share_dict={})
    static particle_item(particle_list, share_dict={})
    static particle_item_list(particle_list)
    rename_params(params, is_particle=True)
decay_cut_ls(decay)
decay_cut_mass(decay)
set_min_max(dic, name, name_min, name_max)

```

extra

```

cal_bins_numbers(self, adapter, data, phsp, read_data, bg=None, bg_weight=None)
cal_chi2(self, read_data=None, bins=[[2, 2], [2, 2], [2, 2]], mass=['R_BD', 'R_CD'])

```

multi_config

```

class MultiConfig(file_names, vm=None, total_same=False, share_dict={}, multi_gpu=False)
    Bases: object
    fit(datas=None, batch=65000, method='BFGS', maxiter=None, print_init_nll=False, callback=None)
    get_all_data()
    get_amplitudes(vm=None)
    get_args_value(bounds_dict)

```

```
get_fcn(datas=None, vm=None, batch=65000)
get_fcns(datas=None, vm=None, batch=65000)
get_params(trainable_only=False)
get_params_error(params=None, datas=None, batch=10000, using_cached=False)
params_trans()
plot_partial_wave(params=None, prefix='figure/all', **kwargs)
reinit_params()
save_params(file_name)
set_params(params, neglect_params=None)
```

particle_function

```
class ParticleFunction(config, name, d_norm=False)
    Bases: object
    amp2s(m)
    density(m)
    mass_linspace(N)
    mass_range()
    phsp_factor(m)
    phsp_fractor(m)
get_particle_function(config, name, d_norm=False)
```

plot

```
class LineStyleSet(file_name, color_first=True)
    Bases: object
    get(id_, default=None)
    get_style(id_)
    save()
build_read_var_function(all_var, where={})
create_chain_property(self, res)
create_plot_var_dic(plot_params)
default_color_generator(color_first)
```

export_legend(ax, filename='legend.pdf', ncol=1)

export legend in Axis ax to file filename

get_chain_property(self, idx, display=True)

Get chain name and curve style in plot

get_chain_property_v1(self, idx, display)

get_chain_property_v2(self, idx, display)

get_dalitz(config, a, b)

get_dalitz_boundary(config, a, b, N=1000)

hist_error(data, bins=50, xrange=None, weights=1.0, kind='poisson')

hist_line(data, weights, bins, xrange=None, inter=1, kind='UnivariateSpline')

interpolate data from histogram into a line

```
>>> import numpy as np
>>> import matplotlib.pyplot
>>> z = np.random.normal(size=1000)
>>> x, y = hist_line(z, None, 50)
>>> a = plt.plot(x, y)
```

hist_line_step(data, weights, bins, xrange=None, inter=1, kind='quadratic')

```
>>> import numpy as np
>>> import matplotlib.pyplot
>>> z = np.random.normal(size=1000)
>>> x, y = hist_line_step(z, None, 50)
>>> a = plt.step(x, y)
```

plot_adaptive_2dpull(config, var1, var2, binning=[[2, 2], [2, 2], [2, 2]], ax=<module 'matplotlib.pyplot' from
'/home/docs/checkouts/readthedocs.org/user_builds/tf-
pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, where={},
cut_zero=True, plot_scatter=True, scatter_style={'c': 'black', 's': 1}, **kwargs)

plot_function_2dpull(data_dict, phsp_dict, bg_dict, var1='x', var2='y', binning=[[2, 2], [2, 2], [2, 2]],
where={}, ax=<module 'matplotlib.pyplot' from
'/home/docs/checkouts/readthedocs.org/user_builds/tf-
pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, cut_zero=True,
plot_scatter=True, scatter_style={'c': 'black', 's': 1}, cmap='jet', **kwargs)

plot_partial_wave(self, params=None, data=None, phsp=None, bg=None, prefix='figure/', res=None,
save_root=False, chains_id_method=None, phsp_rec=None, cut_function=<function
<lambda>>, plot_function=None, **kwargs)

plot partial wave plots

Parameters

- **self** – ConfigLoader object
- **params** – params, dict or FitResults
- **data** – data sample, a list of CalAngleData
- **phsp** – phase space sample, a list of CalAngleData (the same size as data)

- **bg** – background sample, a list of CalAngleData (the same size as data)
- **prefix** – figure saving folder and nameing prefix
- **res** – combination of resonances in partial wave, list of (list of (string for resonances name or int for decay chain index))
- **save_root** – if save weights in a root file, bool
- **chains_id_method** – method of how legend label display, string
- **bin_scale** – more binning in partial waves for a smooth histogram. int
- **batch** – batching in calculating weights, int
- **smooth** – if plot smooth binned kde shape or histogram, bool
- **single_legend** – if save all legend in a file “legend.pdf”, bool
- **plot_pull** – if plot the pull distribution, bool
- **format** – save figure with image format, string (such as “.png”, “.jpeg”)
- **linestyle_file** – legend linestyle configuration file name (YAML format), string (such as “legend.yml”)

plot_partial_wave_interf(*self*, *res1*, *res2*, ***kwargs*)

plotter

class Frame(*var*, *x_range=None*, *nbins=None*, *name=None*, *display=None*, *trans=None*, ***extra*)

Bases: `object`

get_histogram(*data*, *partial=None*, *bin_scale=1*)

set_axis(*axis*, ***config*)

class PlotAllData(*amp*, *data*, *phsp*, *bg=None*, *res=None*, *use_weighted=False*)

Bases: `object`

get_all_histogram(*var*, *bin_scale=3*)

class PlotData(*dataset*, *weight=None*, *partial_weight=None*, *use_weighted=False*)

Bases: `object`

get_histogram(*var*, *partial=None*, ***kwargs*)

total_size()

class PlotDataGroup(*datasets*)

Bases: `object`

get_histogram(*var*, *partial=None*, ***kwargs*)

total_size()

class Plotter(*config*, *legend_file=None*, *res=None*, *datasets=None*, *use_weighted=False*)

Bases: `object`

add_ref_amp(*ref_amp*, *name='reference fit'*)

forzen_style()

get_all_hist(*frame*, *idx=None*, *bin_scale=3*)

create all partial wave histogram for observation frame.

Parameters

- **name** (*Frame*, or *callable*) – Function for get observation in datasets
- **idx** (*int*, *optional*) – data index, None for all data, defaults to None
- **bin_scale** (*float*, *optional*) – smooth bin scale, defaults to 3

Returns

collection of histogram

Return type

dict

get_label(*key*)

get_plot_style(*example_hist*)

get_res_style(*key*)

old_style(*extra_config=None*, *color_first=True*)

context for base style, see matplotlib.rcParams for more configuration

Parameters

- **extra_config** (*dict*, *optional*) – new configs, defaults to None
- **color_first** (*bool*, *optional*) – order of color and linestyle, defaults to True

plot_frame(*name*, *idx=None*, *ax=<module 'matplotlib.pyplot' from
'/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>*, *bin_scale=3*)

plot frame for all partial wave

Parameters

- **name** (*str*) – data variable frame name
- **idx** (*int*, *optional*) – data index, None for all data, defaults to None
- **bin_scale** (*float*, *optional*) – smooth bin scale, defaults to 3
- **ax** (*matplotlib.Axes*, *optional*) – plot on axis ax

Returns

matplotlib.Axes

plot_frame_with_pull(*name*, *idx=None*, *bin_scale=3*, *pull_config=None*)

plot frame with pull for all partial wave

Parameters

- **name** (*str*) – data variable frame name
- **idx** (*int*, *optional*) – data index, None for all data, defaults to None
- **bin_scale** (*float*, *optional*) – smooth bin scale, defaults to 3
- **pull_config** (*dict*, *optional*) – pull plot style, defaults to None

Returns

matplotlib.Axes for plot and pull

plot_var(*frame*, *idx=None*, *ax=<module 'matplotlib.pyplot' from
'/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>*, *bin_scale=3*)

plot data observation for all partial wave

Parameters

- **name** (*Frame*, or *callable*) – Function for get observation in datasets
- **idx** (*int*, *optional*) – data index, None for all data, defaults to None
- **bin_scale** (*float*, *optional*) – smooth bin scale, defaults to 3
- **ax** (*matplotlib.Axes*, *optional*) – plot axis

Returns

save_all_frame(*prefix='figure/'*, *format='png'*, *idx=None*, *plot_pull=False*, *pull_config=None*)

Save all frame in with prefix. like ConfigLoader.plot_partial_waves

Parameters

- **prefix** (*str*, *optional*) – prefix for file name, defaults to “figure/”
- **format** (*str*, *optional*) – figure format, defaults to “png”
- **idx** (*int*, *optional*) – dataset index, defaults to None
- **plot_pull** (*bool*, *optional*) – if plot pulls, defaults to False
- **pull_config** (*dict*, *optional*) – configuration for plot pulls, defaults to None

set_plot_item(*example_hist*)

class ReadData(*var*, *trans=None*)

Bases: *object*

class StyleSet(*file_name*)

Bases: *object*

generate_new_style()

get(*key*, *value=None*)

save()

set(*key*, *value*)

get_all_frame(*self*)

get_all_plotdatas(*self*, *data=None*, *phsp=None*, *bg=None*, *res=None*, *use_weighted=False*)

get_plotter(*self*, *legend_file=None*, *res=None*, *datasets=None*, *use_weighted=False*)

merge_hist(*hists*)

sample

```

class AfterGenerator(gen, f_after=<function AfterGenerator.<lambda>>)
    Bases: BaseGenerator
        cal_max_weight()
        generate(N)
build_phsp_chain(decay_group)
    find common decay those mother particle mass is fixed
build_phsp_chain_sorted(st, final_mi, nodes)
    {A: [B,C, D], R: [B,C]} + {R: M} => ((mr, (mb, mc)), md)
create_cal_calangle(config, include_charge=False)
gen_random_charge(N, random=True)
generate_SDP(config, node, N=1000, include_charge=False, legacy=True)
generate_SDP_p(config, node, N=1000, legacy=False)
generate_phsp(config, N=1000, include_charge=False, cal_max=False)
generate_phsp_p(config, N=1000, cal_max=False)
generate_toy(config, N=1000, force=True, gen=None, gen_p=None, importance_f=None, max_N=100000,
    include_charge=False, cal_phsp_max=False)
    A more accurate method for generating toy data.
    Parameters
        • N – number of events.
        • force – if remove extra data generated.
        • gen – optional function for generate phase space, the return value is same as config.get_data.
        • gen_p – optional function for generate phase space, the return value is dict as {B: pb, C:
            pc, D: pd}.
        • max_N – max number of events for every try.
generate_toy2(config, *args, **kwargs)
generate_toy_o(config, N=1000, force=True, max_N=100000)
generate_toy_p(config, N=1000, force=True, gen_p=None, importance_f=None, max_N=100000,
    include_charge=False, cal_phsp_max=False)
    generate toy data momentum.
get_SDP_generator(config, node, include_charge=False, legacy=True)
get_SDP_p_generator(config, node, legacy=True)
get_SDP_p_generator_legacy(config, node)
get_phsp_generator(config, include_charge=False, nodes=[])
get_phsp_p_generator(config, nodes=[])

```

perfer_node(*struct, index, nodes*)

reorder struct to make node exists in PhaseGenerator

single_sampling(*phsp, amp, N*)

trans_node_order(*struct, index, order_trans, level*)

10.1.4 data_trans

Submodules and Subpackages

dalitz

class Dalitz(*m0, m1, m2, m3*)

Bases: `object`

generate_p(*m12, m23*)

generate monmomentum for dalitz variable

generate_p(*m12, m23, m0, m1, m2, m3*)

generate monmomentum by dalitz variable m12, m23

helicity_angle

class HelicityAngle(*decay_chain*)

Bases: `object`

general implement for angle to monmomentum transform

build_data(*ms, costheta, phi*)

generate monmomentum with M_name = m

cal_angle(*p4*)

eval_phsp_factor(*ms*)

find_variable(*dat*)

generate_p_mass(*name, m, random=False*)

generate monmomentum with M_name = m

get_all_mass(*replace_mass*)

get_mass_range(*name*)

get_phsp_factor(*name, m*)

mass_linspace(*name, N*)

class HelicityAngle1(*decay_chain*)

Bases: `object`

simple implement for angle to monmomentum transform

generate_p(*ms, costheta, phi*)

```

generate_p2(ms, costheta, phi)

generate_p_mass(name, m, random=False)
    generate monmomentum with M_name = m

get_phsp_factor(name, m)

create_rotate_p(ps, ms, costheta, phi)

create_rotate_p_decay(decay_chain, mass, data)

generate_p(ms, msp, costheta, phi)
    ms(0) -> ms(1) + msp(0), costheta(0), phi(0) ms(1) -> ms(2) + msp(1), costheta(1), phi(1) ... ms(n) -> ms(n+1)
    + msp(n), costheta(n), phi(n)

lorentz_neg(pc)

normal(p)

```

10.1.5 experimental

Submodules and Subpackages

build_amp

```

amp_matrix_as_dict(dec, hij)

build_amp2s(dg)

build_amp_matrix(dec, data, weight=None)

build_angle_amp_matrix(dec, data, weight=None)

build_params_vector(dg, data)

build_sum_amplitude(dg, dec_chain, data)

build_sum_angle_amplitude(dg, dec_chain, data)

cached_amp(dg, data, matrix_method=<function build_angle_amp_matrix>)

cached_amp2s(dg, data)

```

extra_amp

extra_data

```

class MultiNpzData(*args, **kwargs)
    Bases: NpzData
    get_data(idx) → list
    get_phsp_noeff()

class NpzData(dic, decay_struct, config=None)
    Bases: SimpleData

```

`get_data(idx) → dict`

`get_particle_p()`

`load_data(files, weights=None, weights_sign=1, charge=None) → dict`

`extra_function`

`extra_function(f0=None, using_numpy=True)`

Using extra function with numerical differentiation.

It can be used for numpy function or numba.vectorize function interface.

```
>>> import numpy as np
>>> sin2 = extra_function(np.sin)
>>> a = tf.Variable([1.0, 2.0], dtype="float64")
>>> with tf.GradientTape(persistent=True) as tape0:
...     with tf.GradientTape(persistent=True) as tape:
...         b = sin2(a)
...         g, = tape.gradient(b, [a,])
...
>>> h, = tape0.gradient(g, [a,])
>>> assert np.allclose(np.sin([1.0, 2.0]), b.numpy())
>>> assert np.allclose(np.cos([1.0, 2.0]), g.numpy())
>>> assert np.sum(np.abs(-np.sin([1.0, 2.0]) - h.numpy())) < 1e-3
```

The numerical accuracy is not so well for second derivative.

`factor_system`

Module for factor system.

`` A = a1 (B x C x D) + a2 (E x F) B = b1 B1 + b2 B2 ``

is a tree structure `` A -> [(a1, [(b1, B1), (b2, B2)], C, D), (a2, E, F)] ``

Each component is a path for root to a leaf. `` (a1, b1), (a1, b2), (a2,) ``

We can add some options to change the possible combination. (TODO)

`flatten_all(x)`

`get_all_chain(a)`

`get_all_partial_amp(amp, data, strip_part=[])`

`get_chain_name(chain)`

`get_id_variable(all_var, var)`

`get_prod_chain(i)`

`get_split_chain(a)`

`partial_amp(amp, data, all_va, need_va)`

`strip_variable(var_all, part=[])`

`temp_var(vm)`

opt_int

```

build_int_matrix(dec, data, weight=None)
build_int_matrix_batch(dec, data, batch=65000)
build_params_matrix(dec)
build_params_vector(dec, concat=True)
build_sum_amplitude(dg, dec_chain, data)
cached_int_mc(dec, data, batch=65000)
gls_combine(fs)
split_gls(dec_chain)

```

wrap_function

```

class Count(idx=0)
    Bases: object
    add(value=1)

class WrapFun(f, jit_compile=False)
    Bases: object

```

10.1.6 generator**Submodules and Subpackages****breit_wigner**

```

class BWGenerator(m0, gamma0, m_min, m_max)
    Bases: BaseGenerator
    DataType
        alias of ndarray
    generate(N)
    integral(x)
    solve(x)

```

generator

class **ARGenerator**(*phsp, amp, max_weight=None*)

Bases: [BaseGenerator](#)

Acceptance-Rejection Sampling

generate(*N*)

class **BaseGenerator**

Bases: [object](#)

DataType = [typing.Any](#)

abstract generate(*N: int*) → [Any](#)

class **GenTest**(*N_max, display=True*)

Bases: [object](#)

add_gen(*n_gen*)

generate(*N*)

set_gen(*n_gen*)

multi_sampling(*phsp, amp, N, max_N=200000, force=True, max_weight=None, importance_f=None, display=True*)

single_sampling2(*phsp, amp, N, max_weight=None, importance_f=None*)

interp_nd

class **InterpND**(*xs, z, indexing='ij'*)

Bases: [object](#)

build_coeffs()

generate(*N*)

integral_step()

class **InterpNDHist**(*xs, z, indexing='ij'*)

Bases: [object](#)

build_coeffs()

generate(*N*)

interp_f(*x*)

integral_step()

linear_interpolation

class LinearInterp(*x, y, epsilon=1e-10*)

Bases: *BaseGenerator*

linear interpolation function for sampling

DataType

alias of ndarray

cal_coeffs()

generate(*N*)

integral(*x*)

solve(*x*)

class LinearInterpImportance(*f, x*)

Bases: *BaseGenerator*

DataType

alias of ndarray

generate(*N*)

interp_sample(*f, xmin, xmax, interp_N, N*)

interp_sample_f(*f, f_interp, N*)

interp_sample_once(*f, f_interp, N, max_rnd*)

sample_test_function(*x*)

plane_2d

class Interp2D(*x, y, z*)

Bases: *TriangleGenerator*

class TriangleGenerator(*x, y, z*)

Bases: *object*

cal_1d_shape()

cal_coeff_left()

cal_coeff_right()

cal_coeffs()

$z = a x + b y + c$

$[x_1, y_1, 1] [a] [z_1] [x_2, y_2, 1] [b] = [z_2] [x_3, y_3, 1] [c] = [z_3]$

$z_c = a x + b y + c$

$s^2 = (x-x_c)^2 + (y-y_c)^2$ $s = 1/b \sqrt{a^2+b^2}(x-x_c)$ $x = b s / \sqrt{a^2+b^2} + x_c$

cal_st_xy(*x, y, bin_index=slice(None, None, None)*)

```

cal_xy_st(s, t, bin_index=slice(None, None, None))

generate(N)

generate_st(N)

solve_left(y, bin_index=slice(None, None, None))

solve_right(y, bin_index=slice(None, None, None))

solve_s(s_r, bin_index=slice(None, None, None))
    int (k_1 s + b_1 )^2 - (k_2 s + b_2)^2 ds = int d s_r

t_min_max(s, bin_index=slice(None, None, None))

```

```

solve_2(a2, a1, x0, y)
    solve (a2 x**2 + a1 x)|_{x0}^{x} = y

solve_3(a3, a2, a1, x0, x_max, y)
    solve (a3 x**3 + a2 x**2 + a1 x**1)|_{x0}^{x} = y

```

square_dalitz_plot

```
class SDPGenerator(m0, mi, legacy=True)
```

Bases: [BaseGenerator](#)

```
generate(N)
```

```

>>> from tf_pwa.generator.square_dalitz_plot import SDPGenerator
>>> gen = SDPGenerator(3.0, [1.0, 0.5, 0.1])
>>> p1, p2, p3 = gen.generate(100)

```

```
generate_SDP(m0, mi, N=1000, legacy=True)
```

generate square dalitz plot ditribution for 1,2

The legacy mode will include a cut off in the threshold.

```
square_dalitz_cut(p)
```

Copy from EvtGen old version

$$|J| = 4pqm_{12} \frac{\partial m_{12}}{\partial m'} \frac{\partial \cos \theta_{12}}{\partial \theta'}$$

$$\frac{\partial m_{12}}{\partial m'} = -\frac{\pi}{2} \sin(\pi m') (m_{12}^{max} - m_{12}^{min})$$

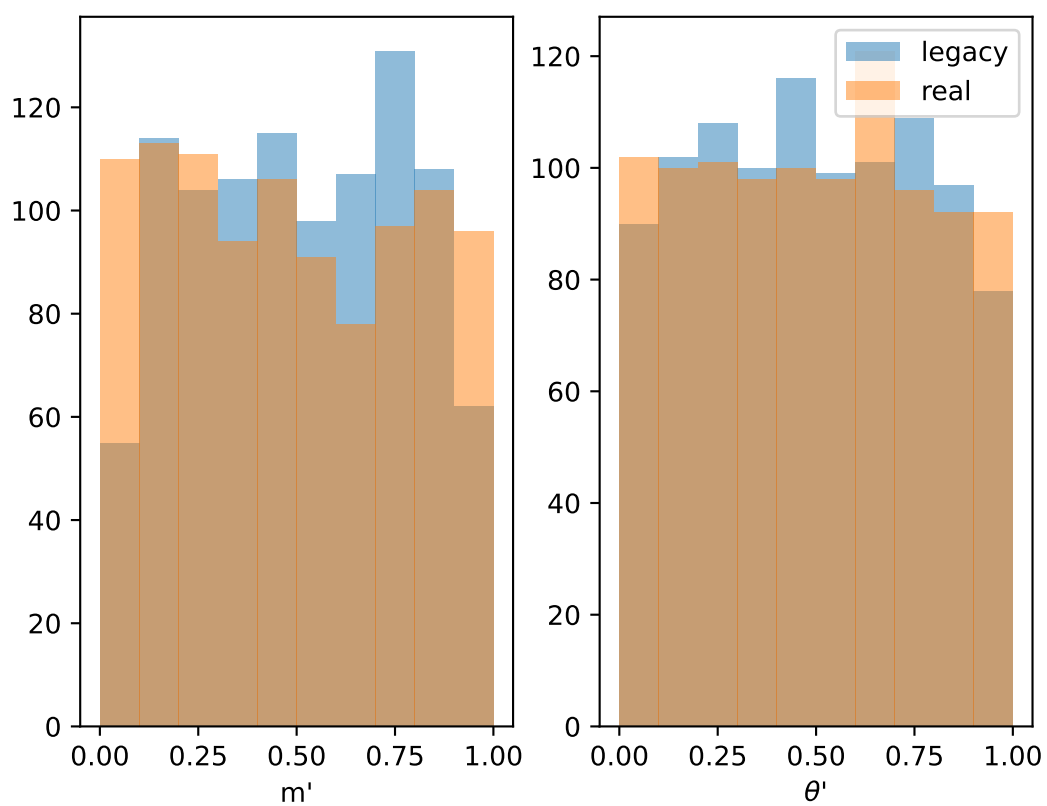
$$\frac{\partial \cos \theta_{12}}{\partial \theta'} = -\pi \sin(\pi \theta')$$

```
square_dalitz_variables(p)
```

Variables used of square dalitz plot, the first 2 is m' and θ' .

$$m' = \frac{1}{\pi} \cos^{-1} \left(2 \frac{m_{12} - m_{12}^{min}}{m_{12}^{max} - m_{12}^{min}} - 1 \right)$$

$$\theta' = \frac{1}{\pi} \theta_{12}$$



10.1.7 model

Submodules and Subpackages

cfit

class ModelCfitExtended(amp, w_bkg=0.001, bg_f=None, eff_f=None)

Bases: [Model](#)

nll(data, mcdata, weight: Tensor = 1.0, batch=None, bg=None, mc_weight=None)

Calculate NLL.

$$\begin{aligned}
 -\ln L &= - \sum_{x_i \in \text{data}} w_i \ln P(x_i; \theta_k) \\
 P(x_i; \theta_k) &= (1 - f_{bg}) \text{Amp}(x_i; \theta_k) + f_{bg} \text{Bg}(x_i; \theta_k) \\
 -\ln L_2 &= -\ln(L \lambda^{N_{\text{data}}} / N_{\text{data}}! e^{-\lambda}) = -L - N_{\text{data}} \ln \lambda + \lambda + C \\
 \lambda &= 1/(1 - f_{bg}) \int \text{Amp}(x_i; \theta_k) d\Phi
 \end{aligned}$$

Parameters

- **data** – Data array
- **mcdata** – MCdata array
- **weight** – Weight of data???
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.
- **bg** – Background data array. It can be set to **None** if there is no such thing.

Returns

Real number. The value of NLL.

nll_grad_batch(data, mcdata, weight, mc_weight)

$$\begin{aligned}
 P &= (1 - \text{frac}) \frac{\text{amp}(\text{data})}{\sum \text{amp}(\text{phsp})} + \text{frac} \frac{\text{bg}(\text{data})}{\sum \text{bg}(\text{phsp})} \\
 nll &= - \sum \log(p) \\
 \frac{\partial nll}{\partial \theta} &= - \sum \frac{1}{p} \frac{\partial p}{\partial \theta} = - \sum \frac{\partial \ln \bar{p}}{\partial \theta} + \frac{\partial nll}{\partial I_{sig}} \frac{\partial I_{sig}}{\partial \theta} + \frac{\partial nll}{\partial I_{bg}} \frac{\partial I_{sig}}{\partial \theta}
 \end{aligned}$$

nll_grad_hessian(data, mcdata, weight=1.0, batch=24000, bg=None, mc_weight=1.0)

The parameters are the same with **self.nll()**, but it will return Hessian as well.

$$\begin{aligned}
 \frac{\partial^2 f}{\partial x_i \partial x_j} &= \frac{\partial y_k}{\partial x_i} \frac{\partial^2 f}{\partial y_k \partial y_l} \frac{\partial y_l}{\partial x_j} + \frac{\partial f}{\partial y_k} \frac{\partial^2 y_k}{\partial x_i \partial x_j} \\
 y &= \{x_i; I_{sig}, I_{bg}\} \\
 \frac{\partial y_k}{\partial x_i} &= (\delta_{ik}; \frac{\partial I_{sig}}{\partial x_i}, \frac{\partial I_{bg}}{\partial x_i})
 \end{aligned}$$

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

class Model_cfit(*amp*, *w_bkg*=0.001, *bg_f*=None, *eff_f*=None, *resolution_size*=1)

Bases: [Model](#)

nll(*data*, *mcddata*, *weight*: *Tensor* = 1.0, *batch*=None, *bg*=None, *mc_weight*=None)

Calculate NLL.

$$-\ln L = - \sum_{x_i \in data} w_i \ln P(x_i; \theta_k)$$

$$P(x_i; \theta_k) = (1 - f_{bg}) Amp(x_i; \theta_k) + f_{bg} Bg(x_i; \theta_k)$$

Parameters

- **data** – Data array
- **mcddata** – MCdata array
- **weight** – Weight of data???
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.
- **bg** – Background data array. It can be set to **None** if there is no such thing.

Returns

Real number. The value of NLL.

nll_grad_batch(*data*, *mcddata*, *weight*, *mc_weight*)

$$P = (1 - frac) \frac{amp(data)}{\sum amp(phsp)} + frac \frac{bg(data)}{\sum bg(phsp)}$$

$$nll = - \sum \log(p)$$

$$\frac{\partial nll}{\partial \theta} = - \sum \frac{1}{p} \frac{\partial p}{\partial \theta} = - \sum \frac{\partial \ln \bar{p}}{\partial \theta} + \frac{\partial nll}{\partial I_{sig}} \frac{\partial I_{sig}}{\partial \theta} + \frac{\partial nll}{\partial I_{bg}} \frac{\partial I_{sig}}{\partial \theta}$$

nll_grad_hessian(*data*, *mcddata*, *weight*=1.0, *batch*=24000, *bg*=None, *mc_weight*=1.0)

The parameters are the same with **self.nll()**, but it will return Hessian as well.

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial y_k}{\partial x_i} \frac{\partial^2 f}{\partial y_k \partial y_l} \frac{\partial y_l}{\partial x_j} + \frac{\partial f}{\partial y_k} \frac{\partial^2 y_k}{\partial x_i \partial x_j}$$

$$y = \{x_i; I_{sig}, I_{bg}\}$$

$$\frac{\partial y_k}{\partial x_i} = (\delta_{ik}; \frac{\partial I_{sig}}{\partial x_i}, \frac{\partial I_{bg}}{\partial x_i})$$

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

```
class Model_cfit_cached(amp, w_bkg=0.001, bg_f=None, eff_f=None)
```

Bases: [Model_cfit](#)

```
nll_grad_batch(data, mcdata, weight, mc_weight)
```

$$P = (1 - \text{frac}) \frac{\text{amp}(\text{data})}{\sum \text{amp}(\text{phsp})} + \text{frac} \frac{\text{bg}(\text{data})}{\sum \text{bg}(\text{phsp})}$$

$$\text{nll} = - \sum \log(p)$$

$$\frac{\partial \text{nll}}{\partial \theta} = - \sum \frac{1}{p} \frac{\partial p}{\partial \theta} = - \sum \frac{\partial \ln \bar{p}}{\partial \theta} + \frac{\partial \text{nll}}{\partial I_{sig}} \frac{\partial I_{sig}}{\partial \theta} + \frac{\partial \text{nll}}{\partial I_{bg}} \frac{\partial I_{bg}}{\partial \theta}$$

```
f_bg(data)
```

```
f_eff(data)
```

custom

```
class BaseCustomModel(amp, w_bkg=1.0, resolution_size=1, extended=False, **kwargs)
```

Bases: [Model](#)

```
eval_nll_part(data, weight=None, norm=None, idx=0)
```

```
eval_normal_factors(mcdata, weight=None)
```

```
nll(data, mcdata, weight: Tensor = 1.0, batch=None, bg=None, mc_weight=1.0)
```

Calculate NLL.

$$-\ln L = - \sum_{x_i \in \text{data}} w_i \ln f(x_i; \theta_k) + \left(\sum w_j \right) \ln \sum_{x_i \in mc} f(x_i; \theta_k)$$

Parameters

- **data** – Data array
- **mcdata** – MCdata array
- **weight** – Weight of data???
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.
- **bg** – Background data array. It can be set to **None** if there is no such thing.

Returns

Real number. The value of NLL.

```
nll_grad_batch(data, mcdata, weight, mc_weight)
```

batch version of `self.nll_grad()`

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in \text{data}} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcdata** –

- **weight** –
- **mc_weight** –

Returns

nll_grad_hessian(*data, mcdata, weight=1.0, batch=24000, bg=None, mc_weight=1.0*)

The parameters are the same with **self.nll()**, but it will return Hessian as well.

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

value_and_grad(*fun*)

class SimpleCFitModel(*amp, w_bkg=1.0, resolution_size=1, extended=False, **kwargs*)

Bases: [BaseCustomModel](#)

eval_nll_part(*data, weight, norm, idx=0*)

eval_normal_factors(*mcdata, weight*)

required_params = ['bg_frac']

class SimpleChi2Model(*amp, w_bkg=1.0, resolution_size=1, extended=False, **kwargs*)

Bases: [BaseCustomModel](#)

fit amp = weight directly. Required set extended = True.

eval_nll_part(*data, weight, norm, idx=0*)

class SimpleClipNllModel(*amp, w_bkg=1.0, resolution_size=1, extended=False, **kwargs*)

Bases: [SimpleNllModel](#)

eval_nll_part(*data, weight, norm, idx=0*)

class SimpleNllFracModel(*amp, w_bkg=1.0, resolution_size=1, extended=False, **kwargs*)

Bases: [BaseCustomModel](#)

eval_nll_part(*data, weight, norm, idx=0*)

eval_normal_factors(*mcdata, weight*)

required_params = ['constr_frac', 'bg_frac']

class SimpleNllModel(*amp, w_bkg=1.0, resolution_size=1, extended=False, **kwargs*)

Bases: [BaseCustomModel](#)

eval_nll_part(*data, weight, norm, idx=0*)

eval_normal_factors(*mcdata, weight*)

model

This module provides methods to calculate NLL(Negative Log-Likelihood) as well as its derivatives.

class BaseModel(*signal, resolution_size=1, extended=False*)

Bases: `object`

This class implements methods to calculate NLL as well as its derivatives for an amplitude model. It may include data for both signal and background.

Parameters

signal – Signal Model

get_params(*trainable_only=False*)

It has interface to `Amplitude.get_params()`.

grad_hessp_batch(*p, data, mcdata, weight, mc_weight*)

`self.nll_grad()` is replaced by this one???

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in data} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcdata** –
- **weight** –
- **mc_weight** –

Returns

nll(*data, mcdata*)

Negative log-Likelihood

nll_grad(*data, mcdata, batch=65000*)

nll_grad_batch(*data, mcdata, weight, mc_weight*)

`self.nll_grad()` is replaced by this one???

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in data} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcdata** –
- **weight** –
- **mc_weight** –

Returns

nll_grad_hessian(*data, mcdata, batch=25000*)

The parameters are the same with `self.nll()`, but it will return Hessian as well.

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

set_params(*var*)

It has interface to `Amplitude.set_params()`.

sum_log_integral_grad_batch(*mcddata*, *ndata*)

sum_nll_grad_bacth(*data*)

sum_resolution(*w*)

property trainable_variables

```
class CombineFCN(model=None, data=None, mcddata=None, bg=None, fcns=None, batch=65000,
                 gauss_constr={})
```

Bases: `object`

This class implements methods to calculate the NLL as well as its derivatives for a general function.

Parameters

- **model** – List of model object.
- **data** – List of data array.
- **mcddata** – list of MCdata array.
- **bg** – list of Background array.
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.

get_grad(*x={}*)

Parameters

x – List. Values of variables.

Return gradients

List of real numbers. The gradients for each variable.

get_grad_hessp(*x*, *p*, *batch*)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

get_nll(*x={}*)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

get_nll_grad(*x*={})

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

get_nll_grad_hessian(*x*={}, *batch*=None)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return hessian

2-D Array of real numbers. The Hessian matrix of the variables.

get_params(*trainable_only*=False)

grad(*x*={})

grad_hessp(*x*, *p*, *batch*=None)

nll_grad(*x*={})

nll_grad_hessian(*x*={}, *batch*=None)

class ConstrainModel(*amp*, *w_bkg*=1.0, *constrain*={})

Bases: [Model](#)

negative log likelihood model with constrains

get_constrain_grad()

constrain: Gauss(mean,sigma)

by add a term $\frac{d}{d\theta_i} \frac{(\theta_i - \bar{\theta}_i)^2}{2\sigma^2} = \frac{\theta_i - \bar{\theta}_i}{\sigma^2}$

get_constrain_hessian()

the constrained parameter's 2nd differentiation

get_constrain_term()

constrain: Gauss(mean,sigma)

by add a term $\frac{(\theta_i - \bar{\theta}_i)^2}{2\sigma^2}$

nll(*data*, *mcdata*, *weight*=1.0, *bg*=None, *batch*=None)

calculate negative log-likelihood

$$-\ln L = - \sum_{x_i \in data} w_i \ln f(x_i; \theta_i) + (\sum w_i) \ln \sum_{x_i \in mc} f(x_i; \theta_i) + cons$$

nll_gradient(*data*, *mcddata*, *weight=1.0*, *batch=None*, *bg=None*)

calculate negative log-likelihood with gradient

$$\frac{\partial}{\partial \theta_i}(-\ln L) = - \sum_{x_i \in \text{data}} w_i \frac{\partial}{\partial \theta_i} \ln f(x_i; \theta_i) + \frac{\sum w_i}{\sum_{x_i \in mc} f(x_i; \theta_i)} \sum_{x_i \in mc} \frac{\partial}{\partial \theta_i} f(x_i; \theta_i) + \text{cons}$$

class FCN(*model*, *data*, *mcddata*, *bg=None*, *batch=65000*, *inmc=None*, *gauss_constr={}*)

Bases: **object**

This class implements methods to calculate the NLL as well as its derivatives for a general function.

Parameters

- **model** – Model object.
- **data** – Data array.
- **mcddata** – MCdata array.
- **bg** – Background array.
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.

get_grad(*x={}*)

Parameters

x – List. Values of variables.

Return gradients

List of real numbers. The gradients for each variable.

get_grad_hessp(*x*, *p*, *batch*)

get_nll(*x={}*)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

get_nll_grad(*x={}*)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

get_nll_grad_hessian(*x={}*, *batch=None*)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return hessian

2-D Array of real numbers. The Hessian matrix of the variables.

get_params(*trainable_only=False*)

grad(*x={}*)

grad_hessp(*x, p, batch=None*)

nll_grad(*x={}*)

nll_grad_hessian(*x={}, batch=None*)

class GaussianConstr(*vm, constraint={}*)

Bases: `object`

get_constrain_grad()

constraint: Gauss(mean,sigma)

by add a term $\frac{d}{d\theta_i} \frac{(\theta_i - \bar{\theta}_i)^2}{2\sigma^2} = \frac{\theta_i - \bar{\theta}_i}{\sigma^2}$

get_constrain_hessian()

the constrained parameter's 2nd differentiation

get_constrain_term()

constraint: Gauss(mean,sigma)

by add a term $\frac{(\theta_i - \bar{\theta}_i)^2}{2\sigma^2}$

update(*constraint={}*)

class MixLogLikelihoodFCN(*model, data, mcdata, bg=None, batch=65000, gauss_constr={}*)

Bases: `CombineFCN`

This class implements methods to calculate the NLL as well as its derivatives for a general function.

Parameters

- **model** – List of model object.
- **data** – List of data array.
- **mcdata** – list of MCdata array.
- **bg** – list of Background array.
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.

get_nll_grad(*x={}*)

Parameters

x – List. Values of variables.

Return nll

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

class Model(*amp*, *w_bkg*=1.0, *resolution_size*=1, *extended*=False, ***kwargs*)

Bases: `object`

This class implements methods to calculate NLL as well as its derivatives for an amplitude model. It may include data for both signal and background.

Parameters

- **amp** – AllAmplitude object. The amplitude model.
- **w_bkg** – Real number. The weight of background.

get_params(*trainable_only*=False)

It has interface to `Amplitude.get_params()`.

get_weight_data(*data*, *weight*=None, *bg*=None, *alpha*=True)

Blend data and background data together multiplied by their weights.

Parameters

- **data** – Data array
- **weight** – Weight for data
- **bg** – Data array for background
- **alpha** – Boolean. If it's true, **weight** will be multiplied by a factor α = ???

Returns

Data, weight. Their length both equals `len(data)+len(bg)`.

grad_hessp_batch(*p*, *data*, *mcddata*, *weight*, *mc_weight*)

`self.nll_grad()` is replaced by this one???

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in data} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcddata** –
- **weight** –
- **mc_weight** –

Returns

mix_data_bakcground(*data*, *bg*)

nll(*data*, *mcddata*, *weight*: *Tensor* = 1.0, *batch*=None, *bg*=None, *mc_weight*=1.0)

Calculate NLL.

$$-\ln L = - \sum_{x_i \in data} w_i \ln f(x_i; \theta_k) + \left(\sum w_j \right) \ln \sum_{x_i \in mc} f(x_i; \theta_k)$$

Parameters

- **data** – Data array
- **mcddata** – MCdata array
- **weight** – Weight of data???

- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.
- **bg** – Background data array. It can be set to **None** if there is no such thing.

Returns

Real number. The value of NLL.

nll_grad(*data*, *mcddata*, *weight*=1.0, *batch*=65000, *bg*=None, *mc_weight*=1.0)

Calculate NLL and its gradients.

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in \text{data}} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in \text{mc}} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in \text{mc}} f(x_i; \theta_k)}$$

The parameters are the same with **self.nll()**, but it will return gradients as well.

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

nll_grad_batch(*data*, *mcddata*, *weight*, *mc_weight*)

batch version of **self.nll_grad()**

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in \text{data}} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in \text{mc}} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in \text{mc}} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcddata** –
- **weight** –
- **mc_weight** –

Returns

nll_grad_hessian(*data*, *mcddata*, *weight*=1.0, *batch*=24000, *bg*=None, *mc_weight*=1.0)

The parameters are the same with **self.nll()**, but it will return Hessian as well.

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

set_params(*var*)

It has interface to **Amplitude.set_params()**.

sum_log_integral_grad_batch(*mcddata*, *ndata*)

sum_nll_grad_bacth(*data*)

sum_resolution(*w*)

class Model_new(*amp, w_bkg=1.0, w_inmc=0, float_wmc=False*)

Bases: [Model](#)

This class implements methods to calculate NLL as well as its derivatives for an amplitude model. It may include data for both signal and background.

Parameters

- **amp** – AllAmplitude object. The amplitude model.
- **w_bkg** – Real number. The weight of background.

get_weight_data(*data, weight=1.0, bg=None, inmc=None, alpha=True*)

Blend data and background data together multiplied by their weights.

Parameters

- **data** – Data array
- **weight** – Weight for data
- **bg** – Data array for background
- **alpha** – Boolean. If it's true, **weight** will be multiplied by a factor $\alpha = ???$

Returns

Data, weight. Their length both equals `len(data)+len(bg)`.

nll(*data, mcdata, weight: Tensor = 1.0, batch=None, bg=None*)

Calculate NLL.

$$-\ln L = - \sum_{x_i \in data} w_i \ln f(x_i; \theta_k) + \left(\sum w_j \right) \ln \sum_{x_i \in mc} f(x_i; \theta_k)$$

Parameters

- **data** – Data array
- **mcdata** – MCdata array
- **weight** – Weight of data???
- **batch** – The length of array to calculate as a vector at a time. How to fold the data array may depend on the GPU computability.
- **bg** – Background data array. It can be set to `None` if there is no such thing.

Returns

Real number. The value of NLL.

nll_grad_batch(*data, mcdata, weight, mc_weight*)

`self.nll_grad_new`

nll_grad_hessian(*data, mcdata, weight, mc_weight*)

The parameters are the same with `self.nll()`, but it will return Hessian as well.

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

clip_log(*x*, *_epsilon=1e-06*)

clip log to allowed large value

get_shape(*x*)

sum_grad_hessp(*f*, *p*, *data*, *var*, *weight=1.0*, *trans=<function identity>*, *resolution_size=1*, *args=()*, *kwargs=None*)

The parameters are the same with **sum_gradient()**, but this function will return hessian as well, which is the matrix of the second-order derivative.

Returns

Real number NLL, list gradient, 2-D list hessian

sum_gradient(*f*, *data*, *var*, *weight=1.0*, *trans=<function identity>*, *resolution_size=1*, *args=()*, *kwargs=None*)

NLL is the sum of $\text{trans}(f(\text{data})) \cdot \text{weight}$; gradient is the derivatives for each variable in `var``.

Parameters

- **f** – Function. The amplitude PDF.
- **data** – Data array
- **var** – List of strings. Names of the trainable variables in the PDF.
- **weight** – Weight factor for each data point. It's either a real number or an array of the same shape with **data**.
- **trans** – Function. Transformation of **data** before multiplied by **weight**.
- **kwargs** – Further arguments for **f**.

Returns

Real number NLL, list gradient

sum_gradient_new(*amp*, *data*, *mcdata*, *weight*, *mcweight*, *var*, *trans=<function log>*, *w_flatmc=<function <lambda>>*, *args=()*, *kwargs=None*)

NLL is the sum of $\text{trans}(f(\text{data})) \cdot \text{weight}$; gradient is the derivatives for each variable in `var``.

Parameters

- **f** – Function. The amplitude PDF.
- **data** – Data array
- **var** – List of strings. Names of the trainable variables in the PDF.
- **weight** – Weight factor for each data point. It's either a real number or an array of the same shape with **data**.
- **trans** – Function. Transformation of **data** before multiplied by **weight**.
- **kwargs** – Further arguments for **f**.

Returns

Real number NLL, list gradient

sum_hessian(*f*, *data*, *var*, *weight=1.0*, *trans=<function identity>*, *resolution_size=1*, *args=()*, *kwargs=None*)

The parameters are the same with **sum_gradient()**, but this function will return hessian as well, which is the matrix of the second-order derivative.

Returns

Real number NLL, list gradient, 2-D list hessian

sum_hessian_new(*amp*, *data*, *mcddata*, *weight*, *mcweight*, *var*, *trans*=<function log>, *w_flatmc*=<function <lambda>>, *args*=(), *kwargs*=None)

The parameters are the same with **sum_gradient()**, but this function will return hessian as well, which is the matrix of the second-order derivative.

Returns

Real number NLL, list gradient, 2-D list hessian

opt_int

class ModelCachedAmp(*amp*, *w_bkg*=1.0)

Bases: [Model](#)

This class implements methods to calculate NLL as well as its derivatives for an amplitude model with Cached Int. It may include data for both signal and background. Cached Int will cause wrong results when float parameters include mass or width.

Parameters

- **amp** – AllAmplitude object. The amplitude model.
- **w_bkg** – Real number. The weight of background.

grad_hessp_batch(*p*, *data*, *mcddata*, *weight*, *mc_weight*)

self.nll_grad() is replaced by this one???

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in data} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcddata** –
- **weight** –
- **mc_weight** –

Returns

nll_grad_batch(*data*, *mcddata*, *weight*, *mc_weight*)

self.nll_grad() is replaced by this one???

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in data} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcddata** –
- **weight** –
- **mc_weight** –

Returns

sum_log_integral_grad_batch(*mcddata*, *ndata*)

`sum_nll_grad_bacth(data)`

class `ModelCachedInt(amp, w_bkg=1.0)`

Bases: `Model`

This class implements methods to calculate NLL as well as its derivatives for an amplitude model with Cached Int. It may include data for both signal and background. Cached Int will cause wrong results when float parameters include mass or width.

Parameters

- **amp** – AllAmplitude object. The amplitude model.
- **w_bkg** – Real number. The weight of background.

`build_cached_int(mcddata, mc_weight, batch=65000)`

`get_cached_int(mc_id)`

`nll_grad_batch(data, mcddata, weight, mc_weight)`

`self.nll_grad()` is replaced by this one???

$$-\frac{\partial \ln L}{\partial \theta_k} = - \sum_{x_i \in \text{data}} w_i \frac{\partial}{\partial \theta_k} \ln f(x_i; \theta_k) + \left(\sum w_j \right) \left(\frac{\partial}{\partial \theta_k} \sum_{x_i \in mc} f(x_i; \theta_k) \right) \frac{1}{\sum_{x_i \in mc} f(x_i; \theta_k)}$$

Parameters

- **data** –
- **mcddata** –
- **weight** –
- **mc_weight** –

Returns

`nll_grad_hessian(data, mcddata, weight=1.0, batch=24000, bg=None, mc_weight=1.0)`

The parameters are the same with `self.nll()`, but it will return Hessian as well.

Return NLL

Real number. The value of NLL.

Return gradients

List of real numbers. The gradients for each variable.

Return Hessian

2-D Array of real numbers. The Hessian matrix of the variables.

`sum_grad_hessp_data2(f, p, var, data, data2, weight=1.0, trans=<function identity>, resolution_size=1, args=(), kwargs=None)`

The parameters are the same with `sum_gradient()`, but this function will return hessian as well, which is the matrix of the second-order derivative.

Returns

Real number NLL, list gradient, 2-D list hessian

`sum_gradient(fs, var, weight=1.0, trans=<function identity>, args=(), kwargs=None)`

NLL is the sum of `trans(f(data))`; `math.*weight`; gradient is the derivatives for each variable in ```var```.

Parameters

- **f** – Function. The amplitude PDF.
- **var** – List of strings. Names of the trainable variables in the PDF.
- **weight** – Weight factor for each data point. It's either a real number or an array of the same shape with **data**.
- **trans** – Function. Transformation of **data** before multiplied by **weight**.
- **kwargs** – Further arguments for **f**.

Returns

Real number NLL, list gradient

sum_gradient_data2(*f, var, data, cached_data, weight=1.0, trans=<function identity>, args=(), kwargs=None*)

NLL is the sum of $\text{trans}(f(\text{data})): \text{math:}^* \text{weight}$; gradient is the derivatives for each variable in ``var`.

Parameters

- **f** – Function. The amplitude PDF.
- **var** – List of strings. Names of the trainable variables in the PDF.
- **weight** – Weight factor for each data point. It's either a real number or an array of the same shape with **data**.
- **trans** – Function. Transformation of **data** before multiplied by **weight**.
- **kwargs** – Further arguments for **f**.

Returns

Real number NLL, list gradient

10.1.8 adaptive_bins

adaptive split data into bins.

class AdaptiveBound(*base_data, bins, base_bound=None*)

Bases: `object`

adaptive bound cut for data value

static base_bound(*data*)

base bound for the data

get_bool_mask(*data*)

bool mask for splitting data

get_bound_patch(***kwargs*)

get_bounds()

get split data bounds

get_bounds_data()

get split data bounds, and the data after splitting

static loop_split_bound(*datas, n, base_bound=None*)

loop for multi_split_bound, so n is list of list of int

static multi_split_bound(*datas*, *n*, *base_bound*=None)

multi data for single_split_bound, so n is list of int

```
>>> data = np.array([[1.0, 2.0, 1.4, 3.1], [2.0, 1.0, 3.0, 1.0]])
>>> bound, _ = AdaptiveBound.multi_split_bound(data, [2, 1])
>>> [(i[0][0]+1e-6, i[1][0]+1e-6) for i in bound]
[(1.0..., 1.7...), (1.7..., 3.1...)]
```

plot_bound(*ax*, ***kwargs*)

static single_split_bound(*data*, *n*=2, *base_bound*=None)

split data in the order of data value

```
>>> data = np.array([1.0, 2.0, 1.4, 3.1])
>>> AdaptiveBound.single_split_bound(data)
[(1.0, 1.7...), (1.7..., 3.1...)]
```

split_data(*data*)

split data, the shape is same as base_data

split_full_data(*data*, *base_index*=None)

split structure data, (TODO because large IO, the method is slow.)

adaptive_shape(*m*, *bins*, *xmin*, *xmax*)

binning_shape_function(*m*, *bins*)

cal_chi2(*numbers*, *n_fp*)

10.1.9 angle

This module implements three classes **Vector3**, **LorentzVector**, **EulerAngle**.

class AlignmentAngle(*alpha*=0.0, *beta*=0.0, *gamma*=0.0)

Bases: [EulerAngle](#)

static angle_px_px(*p1*, *x1*, *p2*, *x2*)

class EulerAngle(*alpha*=0.0, *beta*=0.0, *gamma*=0.0)

Bases: [dict](#)

This class provides methods for Euler angle (α, β, γ)

static angle_zx_z_getx(*z1*, *x1*, *z2*)

The Euler angle from coordinate 1 to coordinate 2. Only the z-axis is provided for coordinate 2, so γ is set to be 0.

Parameters

- **z1** – Vector3 z-axis of the initial coordinate
- **x1** – Vector3 x-axis of the initial coordinate
- **z2** – Vector3 z-axis of the final coordinate

Return euler_angle

EulerAngle object with $\gamma = 0$.

Return x2

Vector3 object, which is the x-axis of the final coordinate when $\gamma = 0$.

static angle_zx_zx(z1, x1, z2, x2)

The Euler angle from coordinate 1 to coordinate 2 (right-hand coordinates).

Parameters

- **z1** – Vector3 z-axis of the initial coordinate
- **x1** – Vector3 x-axis of the initial coordinate
- **z2** – Vector3 z-axis of the final coordinate
- **x2** – Vector3 x-axis of the final coordinate

Returns

Euler Angle object

static angle_zx_zzz_getx(z, x, zi)

The Euler angle from coordinate 1 to coordinate 2. Z-axis of coordinate 2 is the normal vector of a plane.

Parameters

- **z1** – Vector3 z-axis of the initial coordinate
- **x1** – Vector3 x-axis of the initial coordinate
- **z** – list of Vector3 of the plane point.

Return euler_angle

EulerAngle object.

Return x2

list of Vector3 object, which is the x-axis of the final coordinate in zi.

class LorentzVector

Bases: Tensor

This class provides methods for Lorentz vectors (T,X,Y,Z). or -T???

Dot(other)

M()

The invariant mass

M2()

The invariant mass squared

beta()

boost(p)

Boost this Lorentz vector into the frame indicated by the 3-d vector p.

boost_matrix()

boost_vector()

$\beta = (X, Y, Z)/T$:return: 3-d vector β

static from_p4(p_0, p_1, p_2, p_3)

Given **p_0** is a real number, it will make it transform into the same shape with **p_1**.

gamma()

get_T()

get_X()

get_Y()

get_Z()

get_e()

rm???

get_metric()

The metric is (1,-1,-1,-1) by default

neg()

The negative vector

omega()

rest_vector(*other*)

Boost another Lorentz vector into the rest frame of β .

vect()

It returns the 3-d vector (X,Y,Z).

class SU2M(x)

Bases: `dict`

static Boost_z(*omega*)

static Boost_z_from_p(*p*)

static Rotation_y(*beta*)

static Rotation_z(*alpha*)

get_euler_angle()

inv()

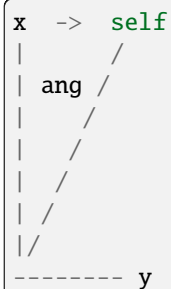
class Vector3

Bases: `Tensor`

This class provides methods for 3-d vectors (X,Y,Z)

angle_from(x, y)

The angle from x-axis providing the x,y axis to define a 3-d coordinate.



Parameters

- **x** – A Vector3 instance as x-axis
- **y** – A Vector3 instance as y-axis. It should be perpendicular to the x-axis.

cos_theta(*other*)

cos theta of included angle

cross(*other*)

Cross product with another Vector3 instance

cross_unit(*other*)

The unit vector of the cross product with another Vector3 object. It has interface to *tf.linalg.normalize()*.

dot(*other*)

Dot product with another Vector3 object

get_X()

get_Y()

get_Z()

norm()

norm2()

The norm square

unit()

The unit vector of itself. It has interface to *tf.linalg.normalize()*.

kine_max(*s12, m0, m1, m2, m3*)

max s23 for s12 in p0 -> p1 p2 p3

kine_min(*s12, m0, m1, m2, m3*)

min s23 for s12 in p0 -> p1 p2 p3

kine_min_max(*s12, m0, m1, m2, m3*)

min max s23 for s12 in p0 -> p1 p2 p3

10.1.10 applications

This module provides functions that implements user-friendly interface to the functions and methods in other modules. It acts like a synthesis of all the other modules of their own physical purposes. In general, users only need to import functions in this module to implement their physical analysis instead of going into every modules. There are some example files where you can figure out how it is used.

cal_hesse_correct(*fcn, params={}, corr_params={}, force_pos=True*)

cal_hesse_error(*fcn, params={}, check_posi_def=True, force_pos=True, save_npy=True*)

This function calculates the errors of all trainable variables. The errors are given by the square root of the diagonal of the inverse Hessian matrix.

Parameters

model – Model.

Return hesse_error

List of errors.

Return inv_he

The inverse Hessian matrix.

cal_significance(*nll1*, *nll2*, *ndf*)

This function calculates the statistical significance.

Parameters

- **nll1** – Float. NLL of the first PDF.
- **nll2** – Float. NLL of the second PDF.
- **ndf** – The difference of the degrees of freedom of the two PDF.

Returns

Float. The statistical significance

compare_result(*value1*, *value2*, *error1*, *error2=None*, *figname=None*, *yrange=None*, *periodic_vars=None*)

Compare two groups of fitting results. If only one error is provided, the figure is $\frac{\mu_1 - \mu_2}{\sigma_1}$; if both errors are provided, the figure is $\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}}$.

Parameters

- **value1** – Dictionary
- **value2** – Dictionary
- **error1** – Dictionary
- **error2** – Dictionary. By default it's None.
- **figname** – String. The output file
- **yrange** – Float. If it's not given, there is no y-axis limit in the figure.
- **periodic_vars** – List of strings. The periodic variables.

Returns

Dictionary of quality figure of each variable.

corr_coef_matrix(*err_mtx*)

This function obtains correlation coefficients matrix of all trainable variables from *.npv file.

Parameters

err_mtx – Array or string (name of the npv file).

Returns

Numpy 2-d array. The correlation coefficient matrix.

fit(*Use='scipy'*, ***kwargs*)

Fit the amplitude model using *scipy*, *iminuit* or *pymultinest*. It imports *fit_scipy*, *fit_minuit*, *fit_multinest* from module **tf_pwa.fit**.

Parameters

- **Use** – String. If it's "scipy", it will call *fit_scipy*; if it's "minuit", it will call *fit_minuit*; if it's "multinest", it will call *fit_multinest*.
- **kwargs** – The arguments will be passed to the three functions above.

For *fit_scipy*

Parameters

- **fcn** – FCN object to be minimized.

- **method** – String. Options in `scipy.optimize`. For now, it implements interface to such as “BFGS”, “L-BFGS-B”, “basinhopping”.
- **bounds_dict** – Dictionary of boundary constrain to variables.
- **kwargs** – Other arguments passed on to `scipy.optimize` functions.

Returns

FitResult object, List of NLLs, List of point arrays.

For `fit_minuit`

Parameters

- **fcn** – FCN object to be minimized.
- **bounds_dict** – Dictionary of boundary constrain to variables.
- **hesse** – Boolean. Whether to call `hesse()` after `migrad()`. It’s True by default.
- **minos** – Boolean. Whether to call `minos()` after `hesse()`. It’s False by default.

Returns

Minuit object

For `fit_multinest` (WIP)

Parameters

fcn – FCN object to be minimized.

fit_fractions(*amp, mcdata, inv_he=None, params=None, batch=25000, res=None, method='old'*)

This function calculate fit fractions of the resonances as well as their coherent pairs. It imports `cal_fitfractions` and `cal_fitfractions_no_grad` from module **tf_pwa.fitfractions**.

$$FF_i = \frac{\int |A_i|^2 d\Omega}{\int |\sum_i A_i|^2 d\Omega} \approx \frac{\sum |A_i|^2}{\sum |\sum_i A_i|^2}$$

gradients???:

$$FF_{i,j} = \frac{\int 2\text{Re}(A_i A_j^*) d\Omega}{\int |\sum_i A_i|^2 d\Omega} = \frac{\int |A_i + A_j|^2 d\Omega}{\int |\sum_i A_i|^2 d\Omega} - FF_i - FF_j$$

hessians:

$$\frac{\partial}{\partial \theta_i} \frac{f(\theta_i)}{g(\theta_i)} = \frac{\partial f(\theta_i)}{\partial \theta_i} \frac{1}{g(\theta_i)} - \frac{\partial g(\theta_i)}{\partial \theta_i} \frac{f(\theta_i)}{g^2(\theta_i)}$$

Parameters

- **amp** – Amplitude object.
- **mcdata** – MCdata array.
- **inv_he** – The inverse of Hessian matrix. If it’s not given, the errors will not be calculated.

Return frac

Dictionary of fit fractions for each resonance.

Return err_frac

Dictionary of their errors. If `inv_he` is None, it will be a dictionary of None.

force_pos_def(*h*)

from pricession lost hessian matrix eigen value is small

$\text{dot}(H, v[:,i]) = e[i] \ v[:,i] \ \text{dot}(H, v[:,i]) = e[i] \ v[:,i] \ \text{dot}(\text{inv}(v), \text{dot}(H, v)) = \text{diag}(e) \ H = \text{dot}(v, \text{dot}(\text{diag}(e), \text{inv}(v)))$

force_pos_def_minuit2(*inv_he*)

force positive defined of error matrix

from minuit2 <https://github.com/root-project/root/blob/master/math/minuit2/sec/MnPosDef.cxx>

gen_data(*amp, Ndata, mcfile, Nbg=0, wbg=0, Poisson_fluc=False, bgfile=None, genfile=None, particles=None*)

This function is used to generate toy data according to an amplitude model.

Parameters

- **amp** – AmplitudeModel???
- **particles** – List of final particles
- **Ndata** – Integer. Number of data
- **mcfile** – String. The MC sample file used to generate signal data.
- **Nbg** – Integer. Number of background. By default it's 0.
- **wbg** – Float. Weight of background. By default it's 0.
- **Poisson_fluc** – Boolean. If it's True, The number of data will be decided by a Poisson distribution around the given value.
- **bgfile** – String. The background sample file used to generate a certain number of background data.
- **genfile** – String. The file to store the generated toy.

Returns

tensorflow.Tensor. The generated toy data.

gen_mc(*mother, daughters, number, outfile=None*)

This function generates phase-space MC data (without considering the effect of detector performance). It imports PhaseSpaceGenerator from module **tf_pwa.phasespace**.

Parameters

- **mother** – Float. The invariant mass of the mother particle.
- **daughters** – List of float. The invariant masses of the daughter particles.
- **number** – Integer. The number of MC data generated.
- **outfile** – String. The file to store the generated MC.

Returns

Numpy array. The generated MC data.

likelihood_profile(*m, var_names, bins=20, minos=True*)

Calculate the likelihood profile for a variable.

Parameters

- **m** – Minuit object
- **var_names** – Either a string or a list of strings
- **bins** – Integer
- **minos** – Boolean. If it's False, the function will call `Minuit.profile()` to derive the 1-d scan of **var_names**; if it's True, the function will call `Minuit.mnprofile()` to derive the likelihood profile, which is much more time-consuming.

Returns

Dictionary indexed by **var_names**. It contains the return of either `Minuit.mnprofile()` or `Minuit.profile()`.

num_hess_inv_3point(*fcn, params={}, _epsilon=0.0005*)

This function calculates the errors of all trainable variables. The errors are given by the square root of the diagonal of the inverse Hessian matrix.

Parameters

model – Model.

Return hesse_error

List of errors.

Return inv_he

The inverse Hessian matrix.

plot_pull(*data, name, nbins=20, norm=False, value=None, error=None*)

This function is used to plot the pull for a data sample.

Parameters

- **data** – List
- **name** – String. Name of the sample
- **nbins** – Integer. Number of bins in the histogram
- **norm** – Boolean. Whether to normalize the histogram
- **value** – Float. Mean value in normalization
- **error** – Float or list. Sigma value(s) in normalization

Returns

The fitted mu, sigma, as well as their errors

10.1.11 breit_wigner

This module provides functions to describe the lineshapes of the intermediate particles, namely generalized Breit-Wigner function. Users can also define new lineshape using the function wrapper **regist_lineshape()**.

BW(*m, m0, g0, *args*)

Breit-Wigner function

$$BW(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0}$$

BWR(*m, m0, g0, q, q0, L, d*)

Relativistic Breit-Wigner function (with running width). It's also set as the default lineshape.

$$BW(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma(m)}$$

BWR2(*m, m0, g0, q2, q02, L, d*)

Relativistic Breit-Wigner function (with running width). Allow complex Γ .

$$BW(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma(m)}$$

BWR_normal($m, m0, g0, q2, q02, L, d$)

Relativistic Breit-Wigner function (with running width) with a normal factor.

$$BW(m) = \frac{\sqrt{m_0 \Gamma(m)}}{m_0^2 - m^2 - i m_0 \Gamma(m)}$$

Bprime($L, q, q0, d$)

Blatt-Weisskopf barrier factors. E.g. the first three orders

L	$B'_L(q, q_0, d)$
0	1
1	$\sqrt{\frac{(q_0 d)^2 + 1}{(q d)^2 + 1}}$
2	$\sqrt{\frac{(q_0 d)^4 + 3 * (q_0 d)^2 + 9}{(q d)^4 + 3 * (q d)^2 + 9}}$

 d is 3.0 by default.**Bprime_num**(L, q, d)

The numerator (as well as the denominator) inside the square root in the barrier factor

Bprime_polynomial(l, z)It stores the Blatt-Weisskopf polynomial up to the fifth order ($L = 5$)**Parameters**

- **l** – The order
- **z** – The variable in the polynomial

Returns

The calculated value

Bprime_q2($L, q2, q02, d$)

Blatt-Weisskopf barrier factors.

GS($m, m0, g0, q, q0, L, d, c_daug2Mass=0.13957039, c_daug3Mass=0.1349768$)**Gamma**($m, gamma0, q, q0, L, m0, d$)

Running width in the RBW

$$\Gamma(m) = \Gamma_0 \left(\frac{q}{q_0} \right)^{2L+1} \frac{m_0}{m} B_L'^2(q, q_0, d)$$

Gamma2($m, gamma0, q2, q02, L, m0, d$)

Running width in the RBW

$$\Gamma(m) = \Gamma_0 \left(\frac{q}{q_0} \right)^{2L+1} \frac{m_0}{m} B_L'^2(q, q_0, d)$$

barrier_factor($l, q, q0, d=3.0, axis=0$)Barrier factor multiplied with q^L , which is used as a combination in the amplitude expressions. The values are cached for L ranging from 0 to **l**.

barrier_factor2(*l, q, q0, d=3.0, axis=-1*)

???

dFun(*s, daug2Mass, daug3Mass*)

dh_dsFun(*s, daug2Mass, daug3Mass*)

fsFun(*s, m2, gam, daug2Mass, daug3Mass*)

get_bprime_coeff(*l*)

The coefficients of polynomial in Bprime function.

$$|\theta_l(jw)|^2 = \sum_{i=0}^l c_i w^{2i}$$

hFun(*s, daug2Mass, daug3Mass*)

one(*args)

A uniform function

regist_lineshape(*name=None*)

It will be used as a wrapper to define various Breit-Wigner functions

Parameters

name – String name of the BW function

Returns

A function used in a wrapper

reverse_bessel_polynomials(*n, x*)

Reverse Bessel polynomials.

$$\theta_n(x) = \sum_{k=0}^n \frac{(n+k)!}{(n-k)!k!} \frac{x^{n-k}}{2^k}$$

to_complex(*i*)

twoBodyCMmom(*m_0, m_1, m_2*)

relative momentum for 0 -> 1 + 2

10.1.12 cal_angle

This module provides functions which are useful when calculating the angular variables.

The full data structure provided is

```
{
  "particle": {
    A: {"p": ..., "m": ...},
    (C, D): {"p": ..., "m": ...},
    ...
  },
  "decay": {
```

(continues on next page)

(continued from previous page)

```

[A->(C, D)+B, (C, D)->C+D]:
{
    A->(C, D)+B: {
        (C, D): {
            "ang": {
                "alpha": [...],
                "beta": [...],
                "gamma": [...]
            },
            "z": [[x1,y1,z1],...],
            "x": [[x2,y2,z2],...]
        },
        B: {...}
    },
    (C, D)->C+D: {
        C: {
            ...,
            "aligned_angle": {
                "alpha": [...],
                "beta": [...],
                "gamma": [...]
            }
        },
        D: {...}
    },
    A->(B, D)+C: {...},
    (B, D)->B+D: {...}
},
...
}

```

Inner nodes are named as tuple of particles.

class CalAngleData

Bases: `dict`

get_angle(*decay*, *p*)

get helicity angle of decay which product particle *p*

get_decay()

get_mass(*name*)

get_momentum(*name*)

get_weight()

mass_hist(*name*, *bins*='sqrt', ***kwargs*)

savetxt(*file_name*, *order*=None, *cp_trans*=False, *save_charge*=False)

Getp(*M_0*, *M_1*, *M_2*)

Consider a two-body decay $M_0 \rightarrow M_1 M_2$. In the rest frame of M_0 , the momentum of M_1 and M_2 are definite.

Parameters

- **M_0** – The invariant mass of M_0
- **M_1** – The invariant mass of M_1
- **M_2** – The invariant mass of M_2

Returns

the momentum of M_1 (or M_2)

Getp2(M_0, M_1, M_2)

Consider a two-body decay $M_0 \rightarrow M_1 M_2$. In the rest frame of M_0 , the momentum of M_1 and M_2 are definite.

Parameters

- **M_0** – The invariant mass of M_0
- **M_1** – The invariant mass of M_1
- **M_2** – The invariant mass of M_2

Returns

the momentum of M_1 (or M_2)

add_mass(*data*: dict, *_decay_chain*: DecayChain | None = None) → dict

add particles mass array for data momentum.

{top:{p:momentum},inner:{p:...},outs:{p:...}} => {top:{p:momentum,m:mass},...}

add_relative_momentum(*data*: dict)

add add rest frame momentum scalar from data momentum.

from {"particle": {A: {"m": ...}, ...}, "decay": {A->B+C: {...}, ...}}

to {"particle": {A: {"m": ...}, ...}, "decay": {[A->B+C,...]: {A->B+C:{...,"|q|": ...},...},...}

add_weight(*data*: dict, *weight*: float = 1.0) → dict

add inner data weights for data.

{...} => {..., "weight": weights}

aligned_angle_ref_rule1(*decay_group*, *decay_chain_struct*, *decay_data*, *data*)

aligned_angle_ref_rule2(*decay_group*, *decay_chain_struct*, *decay_data*, *data*)

cal_angle(*data*, *decay_group*: DecayGroup, *using_topology*=True, *random_z*=True, *r_boost*=True, *final_rest*=True, *align_ref*=None, *only_left_angle*=False)

Calculate helicity angle for particle momentum, add aligned angle.

Params data

dict as {particle: {"p":...}}

Returns

Dictionary of data

cal_angle_from_momentum(*p*, *decs*: DecayGroup, *using_topology*=True, *center_mass*=False, *r_boost*=True, *random_z*=False, *batch*=65000, *align_ref*=None, *only_left_angle*=False) → CalAngleData

Transform 4-momentum data in files for the amplitude model automatically via DecayGroup.

Parameters

- **p** – 4-momentum data
- **decs** – DecayGroup

Returns

Dictionary of data

cal_angle_from_momentum_base(*p*, *decs*: DecayGroup, *using_topology*=True, *center_mass*=False, *r_boost*=True, *random_z*=False, *batch*=65000, *align_ref*=None, *only_left_angle*=False) → CalAngleData

Transform 4-momentum data in files for the amplitude model automatically via DecayGroup.

Parameters

- **p** – 4-momentum data
- **decs** – DecayGroup

Returns

Dictionary of data

cal_angle_from_momentum_id_swap(*p*, *decs*: DecayGroup, *using_topology*=True, *center_mass*=False, *r_boost*=True, *random_z*=False, *batch*=65000, *align_ref*=None, *only_left_angle*=False) → CalAngleData

cal_angle_from_momentum_single(*p*, *decs*: DecayGroup, *using_topology*=True, *center_mass*=False, *r_boost*=True, *random_z*=True, *align_ref*=None, *only_left_angle*=False) → CalAngleData

Transform 4-momentum data in files for the amplitude model automatically via DecayGroup.

Parameters

- **p** – 4-momentum data
- **decs** – DecayGroup

Returns

Dictionary of data

cal_angle_from_particle(*data*, *decay_group*: DecayGroup, *using_topology*=True, *random_z*=True, *r_boost*=True, *final_rest*=True, *align_ref*=None, *only_left_angle*=False)

Calculate helicity angle for particle momentum, add aligned angle.

Params data

dict as {particle: {"p":...}}

Returns

Dictionary of data

cal_chain_boost(*data*, *decay_chain*: DecayChain) → dict

calculate chain boost for a decay chain

cal_helicity_angle(*data*: dict, *decay_chain*: DecayChain, *base_z*=array([0., 0., 1.]), *base_x*=array([1., 0., 0.])) → dict

Calculate helicity angle for A → B + C: θ_B^A, ϕ_B^A from momentum.

from {A: {p: momentum}, B: {p: ...}, C: {p: ...}}

to {A→B+C: {B: {"ang": "alpha": ..., "beta": ..., "gamma": ...}, "x": ..., "z": ...}}

cal_single_boost(*data*, *decay_chain*: DecayChain) → dict

cp_swap_p(*p4*, *finals*, *id_particles*, *cp_particles*)

get_chain_data(*data*, *decay_chain=None*)

get all independent data for a decay chain

get_key_content(*dic*, *key_path*)

get key content. E.g. `get_key_content(data, '/particle/(B, C)/m')`

```
>>> data = {"particle": {"(B, C)": {"p": 0.1, "m": 1.0}, "B": 1.0}}
>>> get_key_content(data, '/particle/(B, C)/m')
1.0
```

get_keys(*dic*, *key_path=""*)

get_keys of nested dictionary

```
>>> a = {"a": 1, "b": {"c": 2}}
>>> get_keys(a)
['/a', '/b/c']
```

get_relative_momentum(*data: dict*, *decay_chain: DecayChain*)

add add rest frame momentum scalar from data momentum.

from {"particle": {A: {"m": ...}, ...}, "decay": {A->B+C: {...}, ...}}

to {"particle": {A: {"m": ...}, ...}, "decay": {A->B+C: {..., "|q|": ...}, ...}}

identical_particles_swap(*id_particles*)

identical_particles_swap_p(*p4*, *id_particles*)

infer_momentum(*data*, *decay_chain: DecayChain*) → dict

infer momentum of all particles in the decay chain from outer particles momentum.

{outs:{p:momentum}} => {top:{p:momentum},inner:{p:...},outs:{p:...}}

parity_trans(*p*, *charges*)

prepare_data_from_dat_file(*fnames*)

[deprecated] angle for amplitude.py

prepare_data_from_dat_file4(*fnames*)

[deprecated] angle for amplitude4.py

prepare_data_from_decay(*fnames*, *decs*, *particles=None*, *dtype=None*, *charges=None*, ***kwargs*)

Transform 4-momentum data in files for the amplitude model automatically via DecayGroup.

Parameters

- **fnames** – File name(s).
- **decs** – DecayGroup
- **particles** – List of Particle. The final particles.
- **dtype** – Data type.

Returns

Dictionary

struct_momentum(*p*, *center_mass=True*) → dict

restructure momentum as dict

{outs:momentum} => {outs:{p:momentum}}

10.1.13 cg

This module provides the function **cg_coef()** to calculate the Clebsch-Gordan coefficients $\langle j_1 m_1 j_2 m_2 | JM \rangle$.

This function has interface to **SymPy** functions if it's installed correctly. Otherwise, it will depend on the input file **tf_pwa/cg_table.json**.

cg_coef(*jb, jc, mb, mc, ja, ma*)

It returns the CG coefficient $\langle j_b m_b j_c m_c | j_a m_a \rangle$, as in a decay from particle *a* to *b* and *c*. It will either call **sympy.physics.quantum.cg()** or **get_cg_coef()**.

get_cg_coef(*j1, j2, m1, m2, j, m*)

If SymPy is not installed, [deprecation] **cg_coef()** will call this function and derive the CG coefficient by searching into **tf_pwa/cg_table.json**.

In fact, **tf_pwa/cg_table.json** only stores some of the coefficients, the others will be obtained by this function using some symmetry properties of the CG table.

Note: **tf_pwa/cg_table.json** only contains the cases where $j_1, j_2 \leq 4$, but this should be enough for most cases in PWA.

10.1.14 config

class ConfigManager

Bases: dict

create_config(*default=None*)

get_config(*name, default=<tf_pwa.config._Default object>*)

get a configuration.

regist_config(*name, var=None*)

regist a configuration.

set_config(*name, var*)

set a configuration.

temp_config(*name, var*)

using_amplitude(*var*)

10.1.15 cov_ten_ir

FL(*s1, s2, s3, S, L*)
FS(*s1, s2, s3, S*)
F_Sigma(*s1, s2, s3, S, L*)
MassiveTransAngle(*p1, p2*)
MasslessTransAngle(*p1, p2*)
NumSL(**args, **kwargs*)
NumSL0(*s1, s2, s3*)
NumSL1(*s1, s2, s3*)
NumSL2(*s1, s2, s3*)
NumSL3(*s1, s2, s3*)
PWFA(*p1, m1_zero, s1, p2, m2_zero, s2, s, S, L*)
SCombLS(*s1, s2, s3, i*)
 LS: i=0 ; i=1 2; i=2 23; i=3 , {S,L}
WFunc1(*s1, s2, s3, S, L*)
WFunc2(*s1, s2, s3, S, L*)
amp0ls(*s1, lens1, s2, lens2, s, lens, theta, phi, S, L*)
cg_in_amp0ls(*s1, lens1, s2, lens2, s, lens, S, L*)
delta_idx_in_amp0ls(*s1, lens1, s2, lens2, s, lens, l*)
force_int(*f*)
ls_selector_weight(*decay, all_ls*)
normal_factor(*L*)
sphericalHarmonic(*l, theta, phi*)
wigerDx(*j, alpha, beta, gamma*)
xyzToangle(*pxyz*)

10.1.16 data

module for describing data process.

All data structure is describing as nested combination of `dict` or `list` for ndarray. Data process is a translation from data structure to another data structure or typical ndarray. Data cache can be implemented based on the dynamic features of `list` and `dict`.

The full data structure is

```

{
  "particle":{
    "A":{"p":...,"m":...}
    ...
  },
  "decay":[
    {
      "A->R1+B": {
        "R1": {
          "ang": {
            "alpha":[...],
            "beta": [...],
            "gamma": [...]
          },
          "z": [[x1,y1,z1],...],
          "x": [[x2,y2,z2],...]
        },
        "B" : {...}
      },
      "R->C+D": {
        "C": {
          ...,
          "aligned_angle":{
            "alpha":[...],
            "beta":[...],
            "gamma":[...]
          }
        },
        "D": {...}
      },
    },
    {
      "A->R2+C": {...},
      "R2->B+D": {...}
    },
    ...
  ],
  "weight": [...]
}

```

```
class EvalLazy(f)
```

```
    Bases: object
```

```
class HeavyCall(f)
```

```
    Bases: object
```

```
class LazyCall(f, x, *args, **kwargs)
```

```
    Bases: object
```

```
    as_dataset(batch=65000)
```

```
    batch(batch, axis=0)
```

```
    copy()
```

```

create_new(f, x, *args, **kwargs)

eval()

get(index, value=None)

get_weight()

merge(*other, axis=0)

set_cached_file(cached_file, name)

class LazyFile(x, *args, **kwargs)
    Bases: LazyCall
    as_dataset(batch=65000)
    create_new(f, x, *args, **kwargs)
    eval()

batch_call(function, data, batch=10000)

batch_call_numpy(function, data, batch=10000)

batch_sum(function, data, batch=10000)

check_nan(data, no_raise=False)
    check if there is nan in data

data_cut(data, expr, var_map=None)
    cut data with boolean expression

    Parameters
    • data – data need to cut
    • expr – cut expression
    • var_map – variable map between parameters in expr and data, [option]

    Returns
    data after being cut,

data_generator(data, fun=<function _data_split>, args=(), kwargs=None, MAX_ITER=1000)
    Data generator: call fun to each data as a generator. The extra arguments will be passed to fun.

data_index(data, key, no_raise=False)
    Indexing data for key or a list of keys.

data_map(data, fun, args=(), kwargs=None)
    Apply fun for each data. It returns the same structure.

data_mask(data, select)
    This function using boolean mask to select data.

    Parameters
    • data – data to select
    • select – 1-d boolean array for selection

```

Returns

data after selection

data_merge(*data, axis=0)

This function merges data with the same structure.

data_replace(data, key, value)

data_shape(data, axis=0, all_list=False)

Get data size.

Parameters

- **data** – Data array
- **axis** – Integer. ???
- **all_list** – Boolean. ???

Returns

data_split(data, batch_size, axis=0)

Split data for batch_size each in axis.

Parameters

- **data** – structured data
- **batch_size** – Integer, data size for each split data
- **axis** – Integer, axis for split, [option]

Returns

a generator for split data

```
>>> data = {"a": [np.array([1.0, 2.0]), np.array([3.0, 4.0])], "b": {"c": np.
↳ array([5.0, 6.0])}, "d": [], "e": {}}
>>> for i, data_i in enumerate(data_split(data, 1)):
...     print(i, data_to_numpy(data_i))
...
0 {'a': [array([1.]), array([3.])], 'b': {'c': array([5.])}, 'd': [], 'e': {}}
1 {'a': [array([2.]), array([4.])], 'b': {'c': array([6.])}, 'd': [], 'e': {}}
```

data_strip(data, keys)

data_struct(data)

get the structure of data, keys and shape

data_to_numpy(dat)

Convert Tensor data to `numpy.ndarray`.

data_to_tensor(dat)

convert data to `tensorflow.Tensor`.

flatten_dict_data(data, fun=<built-in method format of str object>)

Flatten data as dict with structure named as `fun`.

load_dat_file(fnames, particles, dtype=None, split=None, order=None, _force_list=False, mmap_mode=None)

Load *.dat file(s) of 4-momenta of the final particles.

Parameters

- **fnames** – String or list of strings. File names.
- **particles** – List of Particle. Final particles.
- **dtype** – Data type.
- **split** – sizes of each splitted dat files
- **order** – transpose order

Returns

Dictionary of data indexed by Particle.

load_data(*file_name*, ***kwargs*)

Load data file from save_data. The arguments will be passed to `numpy.load()`.

save_data(*file_name*, *obj*, ***kwargs*)

Save structured data to files. The arguments will be passed to `numpy.save()`.

save_datatz(*file_name*, *obj*, ***kwargs*)

Save compressed structured data to files. The arguments will be passed to `numpy.save()`.

set_random_seed(*seed*)

set random seed for random, numpy and tensorflow

split_generator(*data*, *batch_size*, *axis=0*)

Split data for *batch_size* each in *axis*.

Parameters

- **data** – structured data
- **batch_size** – Integer, data size for each split data
- **axis** – Integer, axis for split, [option]

Returns

a generator for split data

```
>>> data = {"a": [np.array([1.0, 2.0]), np.array([3.0, 4.0])], "b": {"c": np.
↳ array([5.0, 6.0])}, "d": [], "e": {}}
>>> for i, data_i in enumerate(data_split(data, 1)):
...     print(i, data_to_numpy(data_i))
...
0 {'a': [array([1.]), array([3.])], 'b': {'c': array([5.])}, 'd': [], 'e': {}}
1 {'a': [array([2.]), array([4.])], 'b': {'c': array([6.])}, 'd': [], 'e': {}}
```

10.1.17 dec_parser

module for parsing decay card *.dec file

do_command(*cmd*, *params*, *lines*)

do command in commands

get_decay(*words*, *lines*)

parser decay command

get_particles(*words*, *_lines*)

parser particles command

get_words(*lines*)

get all words in a lines

load_dec(*s*)

load *.dec string

load_dec_file(*f*)

load *.dec file

process_decay_card(*lines*)

process all the files as a generator

regist_command(*name=None*)

regist command function for command call

remove_comment(*words*)

remove comment string which starts with '#'.

sigle_decay(*s*)

do each decay line

split_lines(*s*)

split each lines

10.1.18 dfun

This module provides functions to calculate the Wigner D-function.

$D_{m_1, m_2}^j(\alpha, \beta, \gamma) = e^{-im_1\alpha} d_{m_1, m_2}^j(\beta) e^{im_2\gamma}$, where the expression of the Wigner d-function is

$$d_{m_1, m_2}^j(\beta) = \sum_{l=0}^{2j} w_l^{(j, m_1, m_2)} \sin^l\left(\frac{\beta}{2}\right) \cos^{2j-l}\left(\frac{\beta}{2}\right),$$

where the weight $w_l^{(j, m_1, m_2)}$ in each term satisfies

$$w_l^{(j, m_1, m_2)} = (-1)^{m_1 - m_2 + k} \frac{\sqrt{(j + m_1)!(j - m_1)!(j + m_2)!(j - m_2)!}}{(j - m_1 - k)!(j + m_2 - k)!(m_1 - m_2 + k)!k!}$$

when $k = \frac{l + m_2 - m_1}{2} \in [\max(0, m_2 - m_1), \min(j - m_1, j + m_2)]$, and otherwise $w_l^{(j, m_1, m_2)} = 0$.

D_matrix_conj(*alpha, beta, gamma, j*)

The conjugated D-matrix element with indices (m_1, m_2) is

$$D_{m_1, m_2}^j(\alpha, \beta, \gamma)^* = e^{im_1\alpha} d_{m_1, m_2}^j(\beta) e^{im_2\gamma}$$

Parameters

- **alpha** – Array
- **beta** – Array
- **gamma** – Array
- **j** – Integer $2j$ in the formula

Returns

Array of the conjugated D-matrices. Same shape as **alpha**, **beta**, and **gamma**

Dfun_delta(*d, ja, la, lb, lc=(0,)*)

The decay from particle *a* to *b* and *c* requires $|l_b - l_c| \leq j$

$$D_{ma,mb-mc} = \delta[(m1, m2) - > (ma, mb, mc)] D_{m1, m2}$$

Dfun_delta_v2(*d, ja, la, lb, lc=(0,)*)

The decay from particle *a* to *b* and *c* requires $|l_b - l_c| \leq j$

$$D_{ma,mb-mc} = \delta[(m1, m2) - > (ma, mb, mc)] D_{m1, m2}$$

delta_D_index(*j, la, lb, lc*)

delta_D_trans(*j, la, lb, lc*)

The decay from particle *a* to *b* and *c* requires $|l_b - l_c| \leq j$

(ja,ja) -> (ja,jb,jc)???

exp_i(*theta, mi*)

$$e^{im\theta}$$

Parameters

- **theta** – Array θ in the formula
- **mi** – Integer or half-integer m in the formula

Returns

Array of tf.complex. Same length as **theta**

get_D_matrix_for_angle(*angle, j, cached=True*)

Interface to *D_matrix_conj()*

Parameters

- **angle** – Dict of angle data {"alpha", "beta", "gamma"}
- **j** – Integer $2j$ in the formula
- **cached** – Haven't been used???

Returns

Array of the conjugated D-matrices. Same length as the angle data

get_D_matrix_lambda(*angle, ja, la, lb, lc=None*)

Get the D-matrix element

Parameters

- **angle** – Dict of angle data {"alpha", "beta", "gamma"}
- **ja** –
- **la** –
- **lb** –
- **lc** –

Returns

small_d_matrix(*theta, j*)

The matrix element of $d^j(\theta)$ is $d_{m_1, m_2}^j(\theta) = \sum_{l=0}^{2j} w_l^{(j, m_1, m_2)} \sin^l(\frac{\theta}{2}) \cos^{2j-l}(\frac{\theta}{2})$

Parameters

- **theta** – Array θ in the formula

- **j** – Integer $2j$ in the formula???

Returns

The d-matrices array. Same length as theta

small_d_weight(j)

For a certain **j**, the weight coefficient with index (m_1, m_2, l) is $w_l^{(j, m_1, m_2)} = (-1)^{m_1 - m_2 + k} \frac{\sqrt{(j+m_1)!(j-m_1)!(j+m_2)!(j-m_2)!}}{(j-m_1-k)!(j+m_2-k)!(m_1-m_2+k)!k!}$, and l is an integer ranging from 0 to $2j$.

Parameters

j – Integer $2j$ in the formula???

Returns

Of the shape **(j + 1, j + 1, j + 1)**. The indices correspond to (l, m_1, m_2)

10.1.19 einsum

class Einsum(*expr, shapes*)

Bases: `object`

einsum(*expr, *args, **kwargs*)

ordered_indices(*expr, shapes*)

find a better order to reduce transpose.

remove_size1(*expr, *args, extra=None*)

remove order independent indices (size 1)

replace_ellipsis(*expr, shapes*)

replace_none_in_shape(*x, num=-1*)

symbol_generate(*base_map*)

tensor_einsum_reduce_sum(*expr, *args, order*)

“abe,bcf->acef” =reshape=> “able1,1bc1f->acef” =product=> “abcfe->acef” =reduce_sum=> “acef”

10.1.20 err_num

class NumberError(*value, error=1.0*)

Bases: `object`

basic class for propagation of error

apply(*fun, grad=None, dx=1e-05*)

property error

exp()

log()

property value

cal_err(*fun, *args, grad=None, dx=1e-05, kwargs=None*)

10.1.21 experimental

10.1.22 fit

class FitResult(*params, fcn, min_nll, ndf=0, success=True, hess_inv=None*)

Bases: `object`

save_as(*file_name, save_hess=False*)

set_error(*error*)

exception LargeNumberError

Bases: `ValueError`

except_result(*fcn, ndf*)

fit_minuit(*fcn, bounds_dict={}, hesse=True, minos=False, **kwargs*)

fit_minuit_v1(*fcn, bounds_dict={}, hesse=True, minos=False, **kwargs*)

Parameters

- **fcn** –
- **bounds_dict** –
- **hesse** –
- **minos** –

Returns

fit_minuit_v2(*fcn, bounds_dict={}, hesse=True, minos=False, **kwargs*)

Parameters

- **fcn** –
- **bounds_dict** –
- **hesse** –
- **minos** –

Returns

fit_multinest(*fcn*)

fit_newton_cg(*fcn, method='Newton-CG', use_hessp=False, check_hess=False, gtol=0.0001*)

fit_root_fitter(*fcn*)

fit_scipy(*fcn, method='BFGS', bounds_dict={}, check_grad=False, improve=False, maxiter=None, jac=True, callback=None, standard_complex=True, grad_scale=1.0, gtol=0.001*)

Parameters

- **fcn** –
- **method** –
- **bounds_dict** –
- **kwargs** –

Returns

10.1.23 fit_improve

class `Cached_FG`(*f_g*, *grad_scale=1.0*)

Bases: `object`

fun(*x*)

grad(*x*)

exception `LineSearchWarning`

Bases: `RuntimeWarning`

class `Seq`(*size=5*)

Bases: `object`

add(*x*)

arg_max()

get_max()

fmin_bfgs_f(*f_g*, *x0*, *B0=None*, *M=2*, *gtol=1e-05*, *Delta=10.0*, *maxiter=None*, *callback=None*, *norm_ord=inf*,
***kwargs*)

test BFGS with nonmonote line search

line_search_nonmonote(*f*, *myfprime*, *xk*, *pk*, *gfk=None*, *old_fval=None*, *fk=None*, *old_old_fval=None*, *args=()*,
c1=0.5, *maxiter=10*)

line_search_wolfe2(*f*, *myfprime*, *xk*, *pk*, *gfk=None*, *fk=None*, *old_fval=None*, *old_old_fval=None*, *args=()*,
c1=0.0001, *c2=0.9*, *amax=None*, *extra_condition=None*, *maxiter=10*)

Find alpha that satisfies strong Wolfe conditions.

Parameters

- **f** (*callable f(x, *args)*) – Objective function.
- **myfprime** (*callable f'(x, *args)*) – Objective function gradient.
- **xk** (*ndarray*) – Starting point.
- **pk** (*ndarray*) – Search direction.
- **gfk** (*ndarray, optional*) – Gradient value for *x=xk* (*xk* being the current parameter estimate). Will be recomputed if omitted.
- **old_fval** (*float, optional*) – Function value for *x=xk*. Will be recomputed if omitted.
- **old_old_fval** (*float, optional*) – Function value for the point preceding *x=xk*
- **args** (*tuple, optional*) – Additional arguments passed to objective function.
- **c1** (*float, optional*) – Parameter for Armijo condition rule.
- **c2** (*float, optional*) – Parameter for curvature condition rule.
- **amax** (*float, optional*) – Maximum step size
- **extra_condition** (*callable, optional*) – A callable of the form `extra_condition(alpha, x, f, g)` returning a boolean. Arguments are the proposed step *alpha* and the corresponding *x*, *f* and *g* values. The line search accepts the value of *alpha* only if this callable returns `True`. If the callable returns `False` for the step length, the algorithm will continue with new iterates. The callable is only called for iterates satisfying the strong Wolfe conditions.

- **maxiter** (*int*, *optional*) – Maximum number of iterations to perform

Returns

- **alpha** (*float or None*) – Alpha for which $\mathbf{x}_{\text{new}} = \mathbf{x}_0 + \text{alpha} * \mathbf{pk}$, or None if the line search algorithm did not converge.
- **fc** (*int*) – Number of function evaluations made.
- **gc** (*int*) – Number of gradient evaluations made.
- **new_fval** (*float or None*) – New function value $f(\mathbf{x}_{\text{new}})=f(\mathbf{x}_0+\text{alpha}*\mathbf{pk})$, or None if the line search algorithm did not converge.
- **old_fval** (*float*) – Old function value $f(\mathbf{x}_0)$.
- **new_slope** (*float or None*) – The local slope along the search direction at the new value $\langle \text{myfprime}(\mathbf{x}_{\text{new}}), \mathbf{pk} \rangle$, or None if the line search algorithm did not converge.

Notes

Uses the line search algorithm to enforce strong Wolfe conditions. See Wright and Nocedal, ‘Numerical Optimization’, 1999, pg. 59-60.

For the zoom phase it uses an algorithm by [...].

minimize(*fun*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *callback=None*, *options=None*)

scalar_search_wolfe2(*phi*, *derphi*, *phi0=None*, *old_phi0=None*, *derphi0=None*, *c1=0.0001*, *c2=0.9*, *amax=None*, *extra_condition=None*, *maxiter=10*)

Find alpha that satisfies strong Wolfe conditions.

alpha > 0 is assumed to be a descent direction.

Parameters

- **phi** (*callable phi(alpha)*) – Objective scalar function.
- **derphi** (*callable phi'(alpha)*) – Objective function derivative. Returns a scalar.
- **phi0** (*float*, *optional*) – Value of phi at 0
- **old_phi0** (*float*, *optional*) – Value of phi at previous point
- **derphi0** (*float*, *optional*) – Value of derphi at 0
- **c1** (*float*, *optional*) – Parameter for Armijo condition rule.
- **c2** (*float*, *optional*) – Parameter for curvature condition rule.
- **amax** (*float*, *optional*) – Maximum step size
- **extra_condition** (*callable*, *optional*) – A callable of the form `extra_condition(alpha, phi_value)` returning a boolean. The line search accepts the value of alpha only if this callable returns True. If the callable returns False for the step length, the algorithm will continue with new iterates. The callable is only called for iterates satisfying the strong Wolfe conditions.
- **maxiter** (*int*, *optional*) – Maximum number of iterations to perform

Returns

- **alpha_star** (*float or None*) – Best alpha, or None if the line search algorithm did not converge.

- **phi_star** (*float*) – phi at alpha_star
- **phi0** (*float*) – phi at 0
- **derphi_star** (*float or None*) – derphi at alpha_star, or None if the line search algorithm did not converge.

Notes

Uses the line search algorithm to enforce strong Wolfe conditions. See Wright and Nocedal, ‘Numerical Optimization’, 1999, pg. 59-60.

For the zoom phase it uses an algorithm by [...].

10.1.24 fitfractions

class FitFractions(*amp, res*)

Bases: `object`

append_int(*mcddata, *args, weight=None, no_grad=False, **kwargs*)

get_frac(*error_matrix=None, sum_diag=True*)

get_frac_diag_sum(*error_matrix=None*)

get_frac_grad(*sum_diag=True*)

init_res_table()

integral(*mcddata, *args, batch=None, no_grad=False, **kwargs*)

cal_fitfractions(*amp, mcddata, res=None, batch=None, args=(), kwargs=None*)

definition:

$$FF_i = \frac{\int |A_i|^2 d\Omega}{\int |\sum_i A_i|^2 d\Omega} \approx \frac{\sum |A_i|^2}{\sum |\sum_i A_i|^2}$$

interference fitfraction:

$$FF_{i,j} = \frac{\int 2\text{Re}(A_i A_j^*) d\Omega}{\int |\sum_i A_i|^2 d\Omega} = \frac{\int |A_i + A_j|^2 d\Omega}{\int |\sum_i A_i|^2 d\Omega} - FF_i - FF_j$$

gradients (for error transfer):

$$\frac{\partial}{\partial \theta_i} \frac{f(\theta_i)}{g(\theta_i)} = \frac{\partial f(\theta_i)}{\partial \theta_i} \frac{1}{g(\theta_i)} - \frac{\partial g(\theta_i)}{\partial \theta_i} \frac{f(\theta_i)}{g^2(\theta_i)}$$

cal_fitfractions_no_grad(*amp, mcddata, res=None, batch=None, args=(), kwargs=None*)

calculate fit fractions without gradients.

eval_integral(*f, data, var, weight=None, args=(), no_grad=False, kwargs=None*)

nll_grad(*f, var, args=(), kwargs=None, options=None*)

sum_gradient(*amp, data, var, weight=1.0, func=<function <lambda>>, grad=True, args=(), kwargs=None*)

sum_no_gradient(*amp, data, var, weight=1.0, func=<function <lambda>>, *, grad=False, args=(), kwargs=None*)

10.1.25 formula

```

BWR_LS_dom(m, m0, g0, thetas, ls, m1, m2, d=3.0, fix_bug1=False)
BWR_coupling_dom(m, m0, g0, l, m1, m2, d=3.0)
BWR_dom(m, m0, g0, l, m1, m2, d=3.0)
BW_dom(m, m0, g0)
Bprime_polynomial(l, z)
create_complex_root_sympy_tfop(f, var, x, x0, epsilon=1e-12, prec=50)
get_relative_p(ma, mb, mc)
get_relative_p2(ma, mb, mc)

```

10.1.26 function

```

nll_funciton(f, data, phsp)
    nagtive log likelihood for minimize

```

10.1.27 gpu_info

```

get_gpu_free_memory(i=0)
get_gpu_info(s)
get_gpu_total_memory(i=0)
get_gpu_used_memory(i=0)

```

10.1.28 histogram

```

class Hist1D(binning, count, error=None)
    Bases: object
    property bin_center
    property bin_width
    chi2()
    draw(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-
        pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, **kwargs)
    draw_bar(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-
        pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, **kwargs)
    draw_error(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-
        pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, fmt='none', **kwargs)

```

```
draw_hist(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, **kwargs)

draw_kde(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, kind='gauss', bin_scale=1.0,
**kwargs)

draw_line(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, num=1000,
kind='UnivariateSpline', **kwargs)

draw_pull(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, **kwargs)

get_bin_weight()

get_count()

static histogram(m, *args, weights=None, mask_error=inf, **kwargs)

ndf()

scale_to(other)

class WeightedData(m, *args, weights=None, **kwargs)
    Bases: Hist1D

    draw_kde(ax=<module 'matplotlib.pyplot' from '/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, kind='gauss', bin_scale=1.0,
**kwargs)

    scale_to(other)

cauchy(x)

epanechnikov(x)

gauss(x)

interp_hist(binning, y, num=1000, kind='UnivariateSpline')
    interpolate data from histogram into a line

plot_hist(binning, count, ax=<module 'matplotlib.pyplot' from
'/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>, **kwargs)

uniform(x)

weighted_kde(m, w, bw, kind='gauss')
```

10.1.29 main

`regist_subcommand(name=None, arg_fun=None, args=None)`

10.1.30 params_trans

```
class ParamsTrans(vm, err_matrix)
    Bases: object
    get_error(vals, keep=False)
    get_error_matrix(vals, keep=False)
    get_grad(val, keep=False)
    mask_params(params)
    trans()
```

10.1.31 particle

This module implements classes to describe particles and decays.

```
class BaseDecay(core, outs, name=None, disable=False, p_break=False, c_break=True, curve_style=None,
                **kwargs)
```

Bases: `object`

Base Decay object

Parameters

- **core** – Particle. The mother particle
- **outs** – List of Particle. The daughter particles
- **name** – String. Name of the decay
- **disable** – Boolean. If it's True???

`as_config()`

`get_id()`

property **name**

```
class BaseParticle(name, J=0, P=-1, C=None, spins=None, mass=None, width=None, id_=None,
                  disable=False, **kwargs)
```

Bases: `object`

Base Particle object. Name is “name[:id]”.

Parameters

- **name** – String. Name of the particle
- **J** – Integer or half-integer. The total spin
- **P** – 1 or -1. The parity

- **spins** – List. The spin quantum numbers. If it's not provided, spins will be `tuple(range(-J, J + 1))`.
- **mass** – Real variable
- **width** – Real variable

add_creator(*d*)

Add a decay reaction where the particle is created.

Parameters

d – BaseDecay object

add_decay(*d*)

Parameters

d – BaseDecay object

as_config()

chain_decay()

return all decay chain self decay to

get_resonances()

return all resonances self decay to

property name

remove_decay(*d*)

Parameters

d – BaseDecay object

set_name(*name*, *id*=None)

class Decay(*core*, *outs*, *name*=None, *disable*=False, *p_break*=False, *c_break*=True, *curve_style*=None, ***kwargs*)

Bases: [*BaseDecay*](#)

General Decay object

barrier_factor(*q*, *q0*)

The cached barrier factors with $d = 3.0$ for all l . For barrier factor, refer to `tf_pwa.breit_wigner.Bprime(L, q, q0, d)`

Parameters

- **q** – Data array
- **q0** – Real number

Returns

1-d array for every data point

generate_params(*name*=None, *_ls*=True)

Generate the name of the variable for every (l,s) pair. In PWA, the variable is usually expressed as g_{ls} .

Parameters

- **name** – String. It is the name of the decay by default
- **_ls** – ???

Returns

List of strings

get_cg_matrix()The matrix indexed by $[(l, s), (\lambda_b, \lambda_c)]$. The matrix element is

$$\sqrt{\frac{2l+1}{2j_a+1}} \langle j_b, j_c, \lambda_b, -\lambda_c | s, \lambda_b - \lambda_c \rangle \langle l, s, 0, \lambda_b - \lambda_c | j_a, \lambda_b - \lambda_c \rangle$$

This is actually the pre-factor of $g_l s$ in the amplitude formula.**Returns**

2-d array of real numbers

get_l_list()

List of l in self.get_ls_list()

get_ls_list()It has interface to `tf_pwa.particle.GetA2BC_LS_list(ja, jb, jc, pa, pb, pc)` :return: List of (l,s) pairs**get_min_l()**

The minimal l in the LS coupling

class DecayChain(chain)Bases: `object`A decay chain. E.g. $A \rightarrow BC, B \rightarrow DE$ **Parameters****chain** – ???**depth_first(node_first=True)**

depth first travel for decay

static from_particles(top, finals)Build possible decay chain Topology. E.g. $a \rightarrow [b,c,d] \Rightarrow [[a \rightarrow rb, r \rightarrow cd], [a \rightarrow rc, r \rightarrow bd], [a \rightarrow rd, r \rightarrow bc]]$ **Parameters**

- **top** – Particle
- **finals** – List of Particle

Returns

DecayChain

static from_sorted_table(decay_dict)Create decay chain from a topology independent structure. E.g. $\{a:[b,c,d], r:[c,d], b:[b], c:[c], d:[d]\} \Rightarrow [a \rightarrow rb, r \rightarrow cd]$ **Parameters****decay_dict** – Dictionary**Returns**

DecayChain

get_all_particles()

get all particles in the decay chains

get_id()

return identity of the decay

get_particle_decay(*particle*)

get particle decay in the chain

sorted_table()

A topology independent structure. E.g. $[a \rightarrow rb, r \rightarrow cd] \Rightarrow \{a:[b,c,d], r:[c,d], b:[b], c:[c], d:[d]\}$

Returns

Dictionary indexed by Particle

sorted_table_layers()

Get the layer of decay chain as sorted table. Or the list of particle with the same number of final particles. So, the first item is always None. E.g. $[a \rightarrow rb, r \rightarrow cd] \Rightarrow [None, [(b, [b])], (c, [c]), (d, [d]), [(r, [c, d])], [(a, [b, c, d])]]$

Returns

List of dictionary

standard_topology()

standard topology structure of the decay chain, all inner particle will be replace as a tuple of out particles. for example $[A \rightarrow R+C, R \rightarrow B+D]$, is $[A \rightarrow (B, D)+C, (B, D) \rightarrow B+D]$

topology_id(*identical=True*)

topology identity

Parameters

identical – allow identical particle in finals

Returns**topology_map(*other=None*)**

Mapping relations of the same topology decay E.g. $[A \rightarrow R+B, R \rightarrow C+D], [A \rightarrow Z+B, Z \rightarrow C+D] \Rightarrow \{A:A, B:B, C:C, D:D, R:Z, A \rightarrow R+B: A \rightarrow Z+B, R \rightarrow C+D: Z \rightarrow C+D\}$

topology_same(*other, identical=True*)

whether self and other is the same topology

Parameters

- **other** – other decay chains
- **identical** – using identical particles

Returns**class DecayGroup(*chains*)**

Bases: `object`

A group of two-body decays.

Parameters

chains – List of DecayChain

as_config()**get_chain_from_particle(*names*)**

get the first decay chain has all particles in names

get_chains_map(*chains=None*)

Parameters

chains –

Returns

get_decay_chain(*id_*)

get decay chain from identity string

get_particle(*name*)

get particle by name

topology_structure(*identical=False, standard=True*)

Parameters

- **identical** –

- **standard** –

Returns

GetA2BC_LS_list(*ja, jb, jc, pa=None, pb=None, pc=None, p_break=False, ca=None*)

The $L - S$ coupling for the decay $A \rightarrow BC$, where L is the orbital angular momentum of B and C , and S is the superposition of their spins. It's required that $|J_B - J_C| \leq S \leq J_B + J_C$ and $|L - S| \leq J_A \leq L + S$. It's also required by the conservation of P parity that L is keep $P_A = P_B P_C (-1)^L$.

Parameters

- **ja** – J of particle A
- **jb** – J of particle B
- **jc** – J of particle C
- **pa** – P of particle A
- **pb** – P of particle B
- **pc** – P of particle C
- **p_break** – allow p violate
- **ca** – enabel c partity select $c=(-1)^{l+s}$

Returns

List of (l, s) pairs.

class ParticleList(*initlist=None*)

Bases: `UserList`

List for Particle

cp_charge_group(*finals, id_p, cp*)

cross_combine(*x*)

Combine every two of a list, as well as give every one of them.??? Can be put to utils.py

Parameters

x –

Returns

simple_cache_fun(*f*)

Parameters

f –

Returns

split_len(*dicts*)

Split a dictionary of lists by their length. E.g. {“b”: [2], “c”: [1, 3], “d”: [1]} => [None, [(“b”, [2]), (‘d’, [1])], [(‘c’, [1, 3])]]

Put to utils.py or _split_len if not used anymore???

Parameters

dicts – Dictionary

Returns

List of dictionary

split_particle_type(*decays*)

split_particle_type_list(*decays*)

Separate initial particle, intermediate particles, final particles in a decay chain.

Parameters

decays – DecayChain

Returns

Set of initial Particle, set of intermediate Particle, set of final Particle

10.1.32 phasespace

class ChainGenerator(*m0, mi*)

Bases: `object`

struct = m0 -> [m1, m2, m3] # (m0, [m1, m2, m3]) m0 -> float mi -> float | struct

cal_max_weight()

generate(*N*)

get_gen(*idx_gen*)

class PhaseSpaceGenerator(*m0, mass*)

Bases: `object`

Phase Space Generator for n-body decay

cal_max_weight()

flatten_mass(*ms, importances=True*)

sampling from mass with weight

generate(*n_iter: int, force=True, flatten=True, importances=True*) → list

generate *n_iter* events

Parameters

- **n_iter** – number of events
- **force** – switch for cutting generated data to required size

- **flatten** – switch for sampling with weights

Returns

daughters 4-momentum, list of ndarray with shape (n_iter, 4)

generate_mass(n_iter)

generate possible inner mass.

generate_momentum(mass, n_iter=None)

generate random momentum from mass, boost them to a same rest frame

generate_momentum_i(m0, m1, m2, n_iter, p_list=[])

|p| = m0, m1, m2 in m0 rest frame :param p_list: extra list for momentum need to boost

get_mass_range()

get_weight(ms, importances=True)

calculate weight of mass

$$w = \frac{1}{w_{max}} \frac{1}{M} \prod_{i=0}^{n-2} q(M_i, M_{i+1}, m_{i+1})$$

mass_importances(mass)

generate possible inner mass.

set_decay(m0, mass)

set decay mass, calculate max weight

$$w_{max} = \frac{1}{M} \prod_{i=0}^{n-2} q(\max(M_i), \min(M_{i+1}), m_{i+1})$$

$$\max(M_i) = M_0 - \sum_{j=1}^i (m_j)$$

$$\min(M_i) = \sum_{j=i}^n (m_j)$$

class UniformGenerator(a, b)

Bases: `object`

generate(N)

generate_phsp(m0, mi, N=1000)

general method to generate decay chain phase sapce >>> (a, b), c = generate_phsp(1.0, (... (0.3, (0.1, 0.1)), ... 0.2), ... N = 10) >>> assert np.allclose(LorentzVector.M(a+b+c), 1.0)

get_p(M, ma, mb)

10.1.33 root_io

load_Ttree(*tree*)

load TTree as dict

load_root_data(*fnames*)

load root file as dict

save_dict_to_root(*dic, file_name, tree_name=None*)

This function stores data arrays in the form of a dictionary into a root file. It provides a convenient interface to uproot.

Parameters

- **dic** – Dictionary of data
- **file_name** – String
- **tree_name** – String. By default it's "tree".

10.1.34 significance

erfc_inverse(*x*)

$\text{erfc-1}(x) = -1/\sqrt{2} * \text{normal_quantile}(0.5 * x)$

normal_quantile(*p*)

Computes quantiles for standard normal distribution $N(0, 1)$ at probability *p*

prob(*chi_2, ndf*)

Computation of the probability for a certain Chi-squared (*chi2*) and number of degrees of freedom (*ndf*).

Calculations are based on the incomplete gamma function $P(a, x)$, where $a = \text{ndf}/2$ and $x = \text{chi}^2/2$.

$P(a, x)$ represents the probability that the observed Chi-squared for a correct model should be less than the value *chi2*.

The returned probability corresponds to $1 - P(a, x)$, which denotes the probability that an observed Chi-squared exceeds the value *chi2* by chance, even for a correct model.

significance(*l1: float, l2: float, ndf: int*) → float

computation of significance for log-likelihood fit values *l1* and *l2* with number of degrees of freedom (*ndf*).

10.1.35 tensorflow_wrapper

class Module

Bases: `object`

numpy_cross(*a, b*)

regist_function(*name, var=None, base_mod=<tf_pwa.tensorflow_wrapper.Module object>*)

set_gpu_mem_growth()

10.1.36 transform

```
class BaseTransform(x: list | str, **kwargs)
    Bases: object
    call(x: Tensor) → Tensor
    inverse(y: Tensor) → Tensor
    read(x: dict) → Tensor

class LinearTrans(x: list | str, k: float = 1.0, b: float = 0.0, **kwargs)
    Bases: BaseTransform
    call(x) → Tensor
    inverse(x: Tensor) → Tensor

create_trans(item: dict) → BaseTransform
```

10.1.37 utils

This module provides some functions that may be useful in other modules.

```
class AttrDict
    Bases: dict

array_split(data, batch=None)
    Split a data array. batch is the number of data in a row.

check_positive_definite(m)
    check if matrix m is postive definite
```

```
>>> check_positive_definite([[1.0,0.0],[0.0, 0.1]])
True
```

```
>>> check_positive_definite([[1.0,0.0],[1.0,-0.1]])
eigvalues: [-0.1 1. ]
False
```

```
create_dir(name)

create_test_config(model_name, params={}, plot_params={})

deep_iter(base, deep=1)

deep_ordered_iter(base, deep=1)

deep_ordered_range(size, deep=1, start=0)

error_print(x, err=None, dig=None)
    It returns a format string "value +/- error". The precision is modified according to err

    Parameters
    • x – Value
    • err – Error
```

Returns

String

fit_normal(data, weights=None)

Fit data distribution with Gaussian distribution. Though minimize the negative log likelihood function

$$-\ln L = \frac{1}{2} \sum w_i \frac{(\mu - x_i)^2}{\sigma^2} + (\sum w_i) \ln(\sqrt{2\pi}\sigma)$$

the fit result can be solved as

$$\frac{\partial(-\ln L)}{\partial \mu} = 0 \Rightarrow \bar{\mu} = \frac{\sum w_i x_i}{\sigma^2 \sum w_i}$$

$$\frac{\partial(-\ln L)}{\partial \sigma} = 0 \Rightarrow \bar{\sigma} = \sqrt{\frac{\sum w_i (\bar{\mu} - x_i)^2}{\sum w_i}}$$

From hessian

$$\frac{\partial^2(-\ln L)}{\partial \mu^2} = \frac{\sum w_i}{\sigma^2}$$

$$\frac{\partial^2(-\ln L)}{\partial \sigma^2} = 3 \sum \frac{\sum w_i (\mu - x)^2}{\sigma^4} - \frac{\sum w_i}{\sigma^2}$$

the error matrix can wrotten as $[[\bar{\sigma}^2/N, 0], [0, \bar{\sigma}^2/(2N)]]$.**flatten_dict_data**(data, fun=<built-in method format of str object>)

Flatten nested dictionary data into one layer dictionary.

Returns

Dictionary

flatten_np_data(data)**is_complex**(x)

If x is of type complex, it returns True.

load_config_file(name)Load config file such as **Resonances.yml**.**Parameters****name** – File name. Either yml file or json file.**Returns**

Dictionary read from the file.

plot_particle_model(model_name, params={}, plot_params={}, axis=None, special_points=None)**pprint**(dicts)

Print dictionary using json format.

print_dic(dic)

Another way to print dictionary.

save_frac_csv(file_name, fit_frac)**std_periodic_var**(p, mid=0.0, pi=3.141592653589793)

Transform a periodic variable into its range.


```
>>> std_periodic_var(math.pi)
-3.1415...
```

```
>>> std_periodic_var(2*math.pi + 0.01)
0.0...
```

Parameters

- **p** – Value
- **mid** – The middle value
- **pi** – Half-range

Returns

The transformed value

std_polar(rho, phi)

To standardize a polar variable. By standard form, it means $\rho > 0$, $-\pi < \phi < \pi$.

Parameters

- **rho** – Real number
- **phi** – Real number

Returns

rho, phi

time_print(f)

It provides a wrapper to print the time cost on a process.

tuple_table(fit_frac, ignore_items=['sum_diag'])

10.1.38 variable

This module implements classes and methods to manage the variables in fitting.

class Bound(a=None, b=None, func=None)

Bases: `object`

This class provides methods to implement the boundary constraint for a variable. It has dependence on `SymPy`. The boundary-transforming function can transform a variable x defined in the real domain to a variable y defined in a limited range (a, b) . y should be the physical parameter but x is the one used while fitting.

Parameters

- **a** – Real number. The lower boundary
- **b** – Real number. The upper boundary
- **func** – String. The boundary-transforming function. By default, if neither **a** or **b** is **None**, **func** is “**(b-a)*(sin(x)+1)/2+a**”; else if only **a** is **None**, **func** is “**b+1-sqrt(x**2+1)**”; else if only **b** is **None**, **func** is “**a-1+sqrt(x**2+1)**”; else **func** is “**x**”.

a, **b**, **func** can be referred by **self.lower**, **self.upper**, **self.func**.

get_d2ydx2(*val*)To calculate the derivative $\frac{dy}{dx}$.**Parameters****val** – Real number x **Returns**Real number $\frac{dy}{dx}$ **get_dydx**(*val*)To calculate the derivative $\frac{dy}{dx}$.**Parameters****val** – Real number x **Returns**Real number $\frac{dy}{dx}$ **get_func**()Initialize the function string into **sympy** objects.**Returns****sympy** objects **f**, **df**, **inv**, which are the function, its derivative and its inverse function.**get_x2y**(*val*)To derive y from x **Parameters****val** – Real number x **Returns**Real number y **get_y2x**(*val*)To derive x from y . y will be set to a if $y < a$, and y will be set to b if $y > b$.**Parameters****val** – Real number y **Returns**Real number x **class SumVar**(*value, grad, var, hess=None*)Bases: **object****from_call**(*var, *args, **kwargs*)**from_call_with_hess**(*var, *args, **kwargs*)**class Variable**(*name, shape=None, cplx=False, vm=None, overwrite=True, is_cp=False, **kwargs*)Bases: **object**

This class has interface to **VarsManager**. It is convenient for users to define a group of real variables, since it may be more perceptually intuitive to define them together.

By calling the instance of this class, it returns the value of this variable. The type is **tf.Tensor**.

Parameters

- **name** – The name of the variable group

- **shape** – The shape of the group. E.g. for a 4*3*2 matrix, **shape** is **[4,3,2]**. By default, **shape** is **[]** for a real variable.
- **cplx** – Boolean. Whether the variable (or the variables) are complex or not.
- **vm** – VarsManager. It is by default the one automatically defined in the global scope by the program.
- **overwrite** – Boolean. If it's True, the program will not throw a warning when overwrite a variable with the same name.
- **kwargs** – Other arguments that may be used when calling **self.real_var()** or **self.cplx_var()**

cplx_cpvar(*polar=True, fix=False, fix_vals=(1.0, 0.0, 0.0, 0.0), value=0.0*)

It implements interface to `VarsManager.add_complex_var()`, but supports variables that are not of non-shape.

Parameters

- **polar** – Boolean. Whether the variable is defined in polar coordinate or in Cartesian coordinate.
- **fix** – Boolean. Whether the variable is fixed. It's enabled only if **self.shape** is **None**.
- **fix_vals** – Length-4 tuple. The value of the fixed complex variable is **fix_vals[0]+fix_vals[1]j**.

cplx_var(*polar=None, fix=False, fix_vals=(1.0, 0.0)*)

It implements interface to `VarsManager.add_complex_var()`, but supports variables that are not of non-shape.

Parameters

- **polar** – Boolean. Whether the variable is defined in polar coordinate or in Cartesian coordinate.
- **fix** – Boolean. Whether the variable is fixed. It's enabled only if **self.shape** is **None**.
- **fix_vals** – Length-2 tuple. The value of the fixed complex variable is **fix_vals[0]+fix_vals[1]j**.

factor_names()

fixed(*value=None*)

Fix this Variable. Note only non-shape real Variable supports this method.

Parameters

value – Real number. The fixed value

freed()

Set free this Variable. Note only non-shape Variable supports this method.

init_name_list()

is_fixed()

r_shareto(*Var*)

Share the radium component to another Variable of the same shape. Only complex Variable supports this method.

Parameters

Var – Variable.

real_var(*value=None, range_=None, fix=False*)

It implements interface to `VarsManager.add_real_var()`, but supports variables that are not of non-shape.

Parameters

- **value** – Real number. The value of all real components.
- **range** – Length-2 array. The length of all real components.
- **fix** – Boolean. Whether the variable is fixed.

rename(*new_name*)

Rename this Variable.

sameas(*Var*)

Set the Variable to be the same with another Variable of the same shape.

Parameters

Var – Variable.

set_bound(*bound, func=None, overwrite=False*)

Set boundary for this Variable. Note only non-shape real Variable supports this method.

Parameters

- **bound** – Length-2 tuple.
- **func** – String. Refer to class `tf_pwa.variable.Bound`.
- **overwrite** – Boolean. If it's `True`, the program will not throw a warning when overwrite a variable with the same name.

set_fix_idx(*fix_idx=None, fix_vals=None, free_idx=None*)

Parameters

- **fix_idx** – Integer or list of integers. Which complex component in the innermost layer of the variable is fixed. E.g. If `self.shape==[2,3,4]` and `fix_idx==[1,2]`, then `Variable()[i][j][1]` and `Variable()[i][j][2]` will be the fixed value.
- **fix_vals** – Float or length-2 float list for complex variable. The fixed value.
- **free_idx** – Integer or list of integers. Which complex component in the innermost layer of the variable is set free. E.g. If `self.shape==[2,3,4]` and `fix_idx==[0]`, then `Variable()[i][j][0]` will be set free.

set_phi(*phi, index=None*)

set_rho(*rho, index=None*)

set_same_ratio()

set_value(*value, index=None*)

property value

Ndarray of `self.shape`.

Type

return

property variables

Names of the real variables contained in this Variable instance.

Returns

List of string.

class VarsManager(*name=""*, *dtype=tf.float64*, *multi_gpu=None*)

Bases: `object`

This class provides methods to operate the variables in fitting. Every variable is a 1-d **tf.Variable** of **dtype** (**tf.float64** by default).

All variables are stored in a dictionary **self.variables**. The indices of the dictionary are the variables' names, so name property in **tf.Variable** does not matter. All methods intended to change the variables are operating **self.variables** directly.

Besides, all trainable variables' names will be stored in a list **self.trainable_vars**.

add_cartesiancp_var(*name*, *polar=None*, *trainable=True*, *fix_vals=(1.0, 0.0, 0.0, 0.0)*)

Add a complex variable. Two real variables named **name+'r'** and **name+'i'** will be added into **self.variables**. The initial values will be given automatically according to its form of coordinate.

Parameters

- **name** – The name of the complex variable.
- **polar** – Boolean. If it's **True**, **name+'r'** and **name+'i'** are defined in polar coordinate; otherwise they are defined in Cartesian coordinate.
- **trainable** – Boolean. If it's **True**, real variables **name+'r'** and **name+'i'** will be trainable.
- **fix_vals** – Length-4 array. If **trainable=False**, the fixed values for **name+'r'** and **name+'i'** are **fix_vals[0]**, **fix_vals[1]** respectively.

add_complex_var(*name*, *polar=None*, *trainable=True*, *fix_vals=(1.0, 0.0)*)

Add a complex variable. Two real variables named **name+'r'** and **name+'i'** will be added into **self.variables**. The initial values will be given automatically according to its form of coordinate.

Parameters

- **name** – The name of the complex variable.
- **polar** – Boolean. If it's **True**, **name+'r'** and **name+'i'** are defined in polar coordinate; otherwise they are defined in Cartesian coordinate.
- **trainable** – Boolean. If it's **True**, real variables **name+'r'** and **name+'i'** will be trainable.
- **fix_vals** – Length-2 array. If **trainable=False**, the fixed values for **name+'r'** and **name+'i'** are **fix_vals[0]**, **fix_vals[1]** respectively.

add_real_var(*name*, *value=None*, *range_=None*, *trainable=True*)

Add a real variable named **name** into **self.variables**. If **value** and **range_** are not provided, the initial value is set to be a uniform random number between 0 and 1.

Parameters

- **name** – The name of the variable, the index of this variable in **self.variables**
- **value** – The initial value.
- **range** – Length-2 array. It's useless if **value** is given. Otherwise the initial value is set to be a uniform random number between **range_[0]** and **range_[0]**.

- **trainable** – Boolean. If it's **True**, the variable is trainable while fitting.

batch_sum_var(*fun, data, batch=65000*)

error_trans(*err_matrix*)

get(*name, val_in_fit=True*)

Get a real variable. If `val_in_fit` is **True**, this is the variable used in fitting, not considering its boundary transformation.

Parameters

name – String

Returns

tf.Variable

get_all_dic(*trainable_only=False*)

Get a dictionary of all variables.

Parameters

trainable_only – Boolean. If it's **True**, the dictionary only contains the trainable variables.

Returns

Dictionary

get_all_val(*val_in_fit=False*)

Get the values of all trainable variables.

Parameters

val_in_fit – Boolean. If it's **True**, the values will be the ones that are actually used in fitting (thus may not be the physical values because of the boundary transformation).

Returns

List of real numbers.

mask_params(*params*)

minimize(*fcn, jac=True, method='BFGS', mini_kwargs={}*)

minimize a give function

minimize_error(*fcn, fit_result*)

read(*name*)

refresh_vars(*init_val=None, bound_dic=None*)

Refresh all trainable variables

remove_bound()

Remove a boundary for a variable

remove_var(*name*)

Remove a variable from **self.variables**. More specifically, two variables (**name+'r'** and **name+'i'**) will be removed if it's complex.

Parameters

name – The name of the variable

rename_var(*name, new_name, cplx=False*)

Rename a variable.

Parameters

- **name** – Name of the variable
- **new_name** – New name
- **cplx** – Boolean. Users should indicate if this variable is complex or not.

rp2xy(*name*)

Transform a complex variable into Cartesian coordinate. :param name: String

rp2xy_all(*name_list=None*)

If **name_list** is not provided, this method will transform all complex variables into Cartesian coordinate.

Parameters

name_list – List of names of complex variables

set(*name, value, val_in_fit=True*)

Set value for a real variable. If **val_in_fit** is **True**, this is the variable used in fitting, not considering its boundary transformation.

Parameters

- **name** – String
- **value** – Real number

set_all(*vals, val_in_fit=False*)

Set values for some variables.

Parameters

vals – It can be either a dictionary or a list of real numbers. If it's a list, the values correspond to all trainable variables in order.

set_bound(*bound_dic, func=None, overwrite=False*)

Set boundary for the trainable variables. The variables will be constrained in their ranges while fitting.

Parameters

- **bound_dic** – Dictionary. E.g. {"name1":(-1.0,1.0), "name2":(None,1.0)}. In this example, **None** means it has no lower limit.
- **func** – String. Users can provide a string to describe the transforming function. For details, refer to class **tf_pwa.variable.Bound**.
- **overwrite** – Boolean. If it's **True**, the program will not throw a warning when overwrite a variable with the same name.

set_fix(*name, value=None, unfix=False*)

Fix or unfix a variable (change the trainability) :param name: The name of the variable :param value: The fixed value. It's useless if **unfix=True**. :param unfix: Boolean. If it's **True**, the variable will become trainable rather than be fixed.

set_same(*name_list, cplx=False*)

Set some variables to be the same.

Parameters

- **name_list** – List of strings. Name of the variables.
- **cplx** – Boolean. Whether the variables are complex or real.

set_share_r(*name_list*)

If some complex variables want to share their radia variable while their phase variable are still different. Users can set this type of constrain using this method.

Parameters

name_list – List of strings. Note the strings should be the name of the complex variables rather than of their radium parts.

set_trans_var(*xvals*)

$$y = y(x)$$

Parameters

fcn_grad – The return of class **tf_pwa.model???**

Returns

standard_complex()

std_polar(*name*)

Transform a complex variable into standard polar coordinate, which mean its radium part is positive, and its phase part is between $-\pi$ to π . :param name: String

std_polar_all()

Transform all complex variables into standard polar coordinate.

temp_params(*params*)

property trainable_variables

List of tf.Variable. It is similar to **tf.keras.Model.trainable_variables**.

trans_error_matrix(*hess_inv*, *xvals*)

Bound trans for error matrix $F(x) = F(y(x))$, $V_y = y' V_x y'$

Returns

trans_f_grad_hess(*f*)

$$F(x) = F(y(x)), G(x) = \frac{dF}{dx} = \frac{dF}{dy} \frac{dy}{dx}$$

Parameters

fcn_grad – The return of class **tf_pwa.model???**

Returns

trans_fcn_grad(*fcn_grad*)

$$F(x) = F(y(x)), G(x) = \frac{dF}{dx} = \frac{dF}{dy} \frac{dy}{dx}$$

Parameters

fcn_grad – The return of class **tf_pwa.model???**

Returns

trans_grad_hessp(*f*)

$$F(x) = F(y(x)), G(x) = \frac{dF}{dx} = \frac{dF}{dy} \frac{dy}{dx}$$

Parameters

fcn_grad – The return of class **tf_pwa.model???**

Returns

trans_params(*polar*)

Transform all complex variables into either polar coordinate or Cartesian coordinate.

Parameters

polar – Boolean

xy2rp(*name*)

Transform a complex variable into polar coordinate. :param name: String

xy2rp_all(*name_list=None*)

If **name_list** is not provided, this method will transform all complex variables into polar coordinate.

Parameters

name_list – List of names of complex variables

combineVM(*vm1, vm2, name="", same_list=None*)

This function combines two VarsManager objects into one. (WIP)

Parameters

- **name** – The name of this combined VarsManager
- **same_list** – To make some variables in the two VarsManager to be the same. E.g. if `same_list = ["var", ["var1", "var2"]]`, then “var” in vm1 and vm2 will be the same, and “var1” in vm1 and “var2” in vm2 will be the same.

deep_stack(*dic, deep=1*)

10.1.39 version

10.1.40 vis

class DotGenerator(*top*)

Bases: `object`

static dot_chain(*chains, has_label=True*)

dot_default_edge = ' "{}" -> "{}";\n'

dot_default_node = ' "{}" [shape=none];\n'

dot_head = '\ndigraph {\n rankdir=LR;\n node [shape=point];\n edge [arrowhead=none, labelfloat=true];\n'

dot_label_edge = ' "{}" -> "{}" [label="{}"];\n'

dot_ranksame = ' {{ rank=same {} }};\n'

dot_tail = '}\n'

get_dot_source()

draw_decay_struct(*decay_chain, show=False, **kwargs*)

get_decay_layout(*decay_chain*)

get_layout(*decay_chain, xs, ys*)

get_node_layout(*decay_chain*)

plot_decay_struct(*decay_chain, ax=<module 'matplotlib.pyplot' from
'/home/docs/checkouts/readthedocs.org/user_builds/tf-pwa/envs/stable/lib/python3.10/site-packages/matplotlib/pyplot.py'>*)

reorder_final_particle(*decay_chain, ys*)

10.1.41 `weight_smear`

`dirichlet_smear(weight, **kwargs)`

`gamma_smear(weight, **kwargs)`

`get_weight_smear(name)`

`poisson_smear(weight, **kwargs)`

`register_weight_smear(name)`

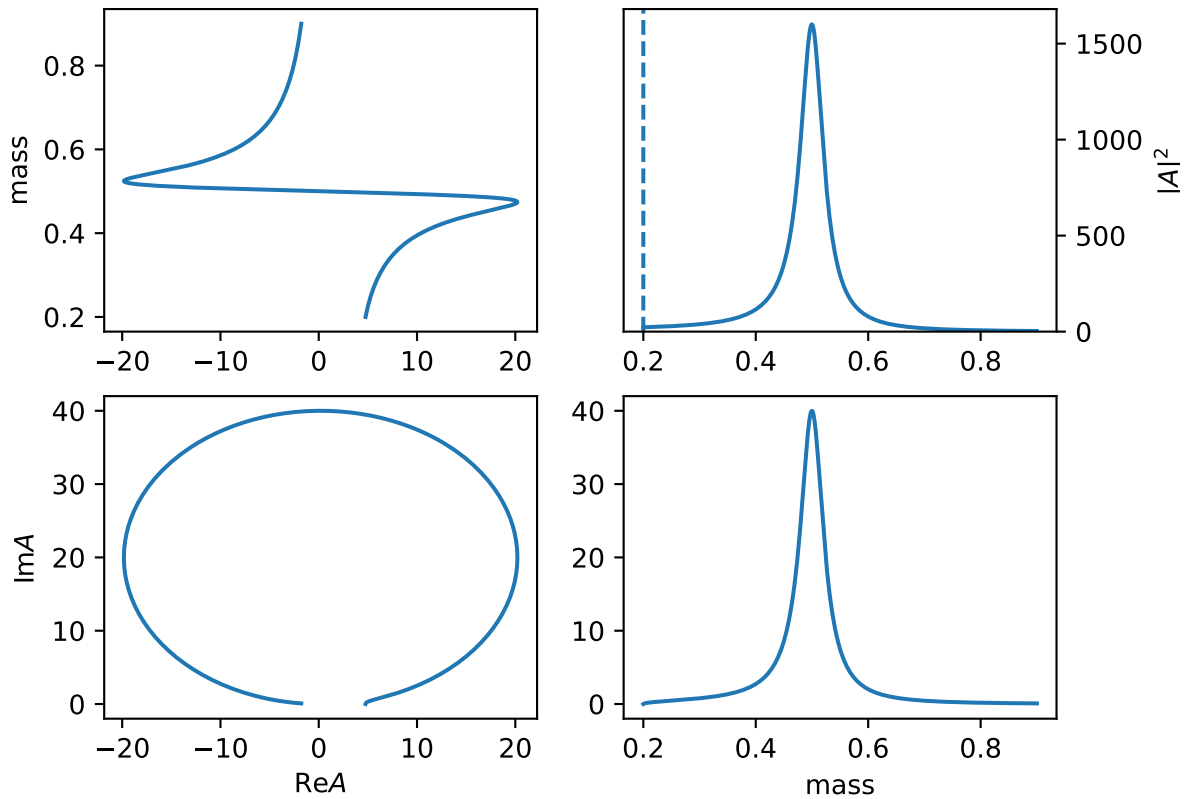
See also:

- `modindex`
- `genindex`

AVAILABLE RESONANCES MODEL

1. "default", "BWR" (*Particle*)

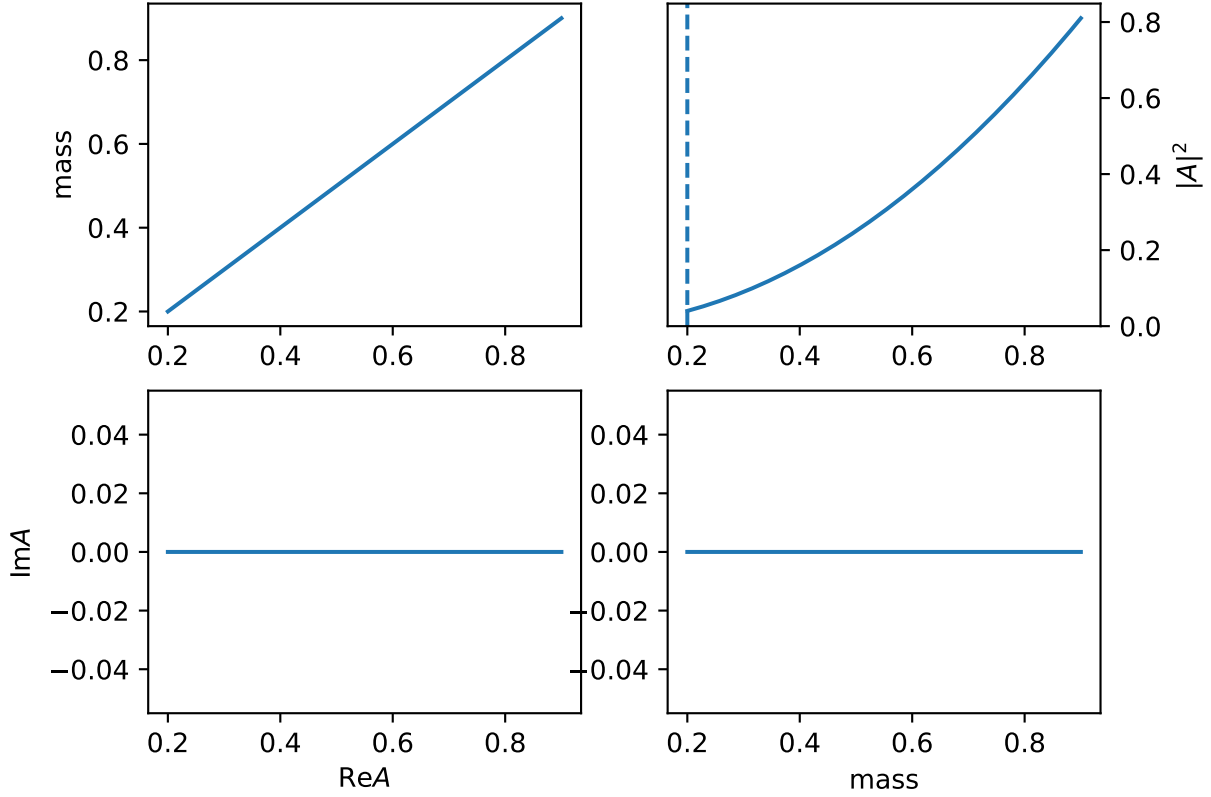
$$R(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma(m)}$$



2. "x" (*ParticleX*)

simple particle model for mass, (used in expr)

$$R(m) = m$$



3. "BWR2" (*ParticleBWR2*)

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

The difference of BWR, BWR2 is the behavior when mass is below the threshold ($m_0 = 0.1 < 0.1 + 0.1 = m_1 + m_2$).

4. "BWR_below" (*ParticleBWRBelowThreshold*)

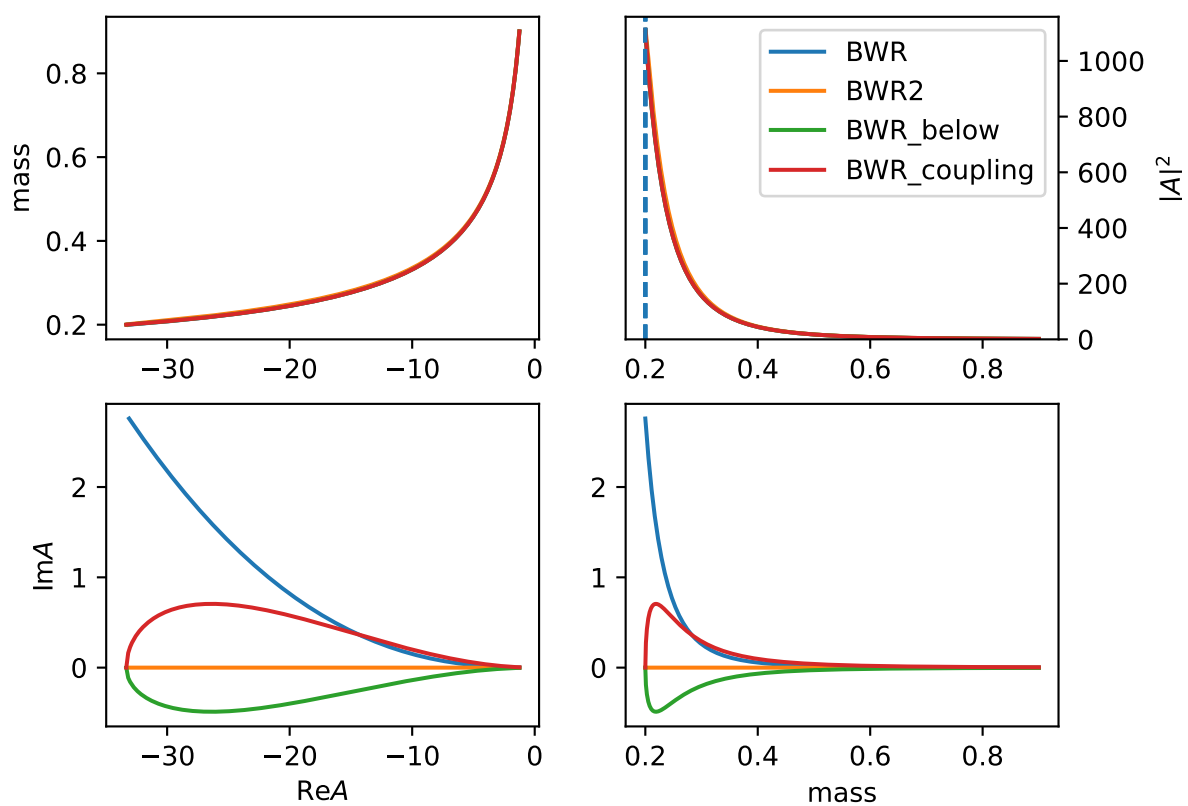
$$R(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma(m)}$$

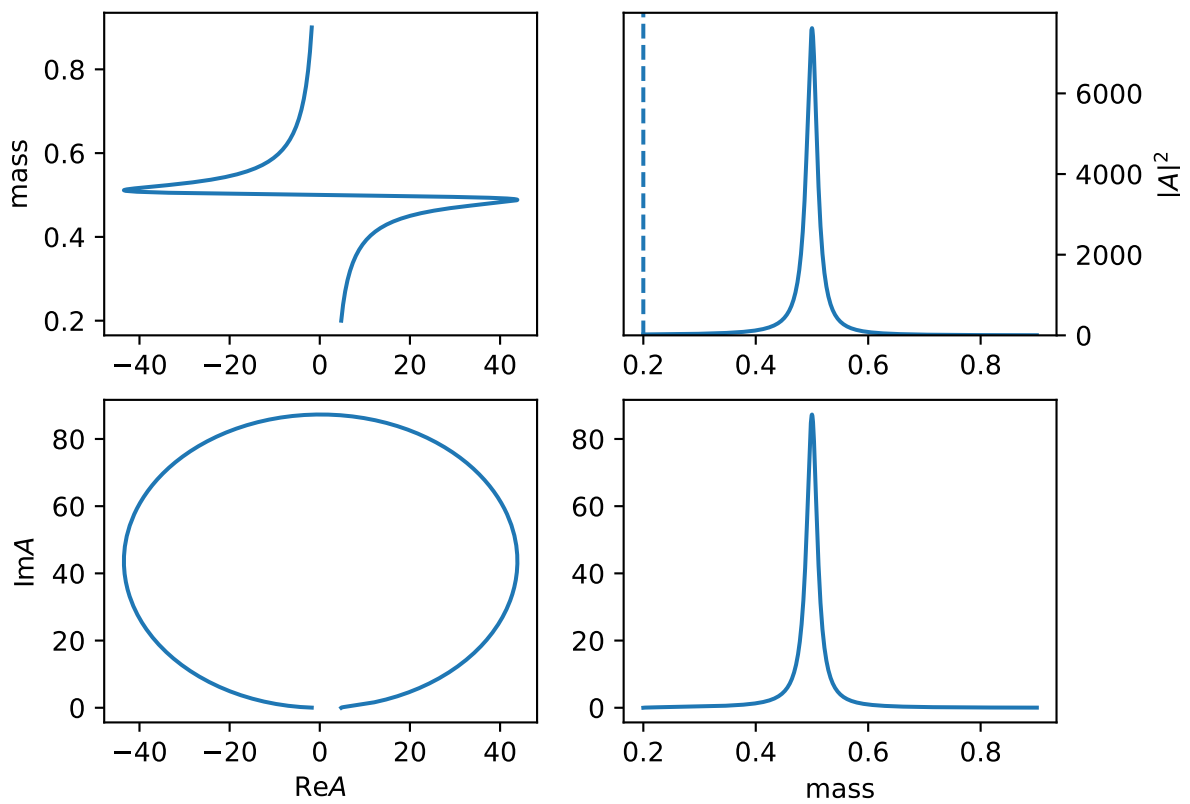
5. "BWR_coupling" (*ParticleBWRCoupling*)

Force $q_0 = 1/d$ to avoid below threshold condition for BWR model, and remove other constant parts, then the Γ_0 is coupling parameters.

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0 \frac{q}{m} q^{2l} B_L'^2(q, 1/d, d)}$$

6. "BWR_normal" (*ParticleBWR_normal*)





$$R(m) = \frac{\sqrt{m_0 \Gamma(m)}}{m_0^2 - m^2 - im_0 \Gamma(m)}$$

7. "GS_rho" (*ParticleGS*)

Gounaris G.J., Sakurai J.J., Phys. Rev. Lett., 21 (1968), pp. 244-247

c_daug2Mass: mass for daughter particle 2 (π^+) 0.13957039

c_daug3Mass: mass for daughter particle 3 (π^0) 0.1349768

$$R(m) = \frac{1 + D\Gamma_0/m_0}{(m_0^2 - m^2) + f(m) - im_0\Gamma(m)}$$

$$f(m) = \Gamma_0 \frac{m_0^2}{q_0^3} \left[q^2 [h(m) - h(m_0)] + (m_0^2 - m^2) q_0^2 \frac{dh}{dm} \Big|_{m_0} \right]$$

$$h(m) = \frac{2}{\pi} \frac{q}{m} \ln \left(\frac{m + 2q}{2m_\pi} \right)$$

$$\frac{dh}{dm} \Big|_{m_0} = h(m_0) [(8q_0^2)^{-1} - (2m_0^2)^{-1}] + (2\pi m_0^2)^{-1}$$

$$D = \frac{f(0)}{\Gamma_0 m_0} = \frac{3}{\pi} \frac{m_\pi^2}{q_0^2} \ln \left(\frac{m_0 + 2q_0}{2m_\pi} \right) + \frac{m_0}{2\pi q_0} - \frac{m_\pi^2 m_0}{\pi q_0^3}$$

8. "BW" (*ParticleBW*)

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0}$$

9. "LASS" (*ParticleLass*)

$$R(m) = \frac{m}{q \cot \delta_B - iq} + e^{2i\delta_B} \frac{m_0 \Gamma_0 \frac{m_0}{q_0}}{(m_0^2 - m^2) - im_0 \Gamma_0 \frac{q}{m} \frac{m_0}{q_0}}$$

$$\cot \delta_B = \frac{1}{aq} + \frac{1}{2} r q$$

$$e^{2i\delta_B} = \cos 2\delta_B + i \sin 2\delta_B = \frac{\cot^2 \delta_B - 1}{\cot^2 \delta_B + 1} + i \frac{2 \cot \delta_B}{\cot^2 \delta_B + 1}$$

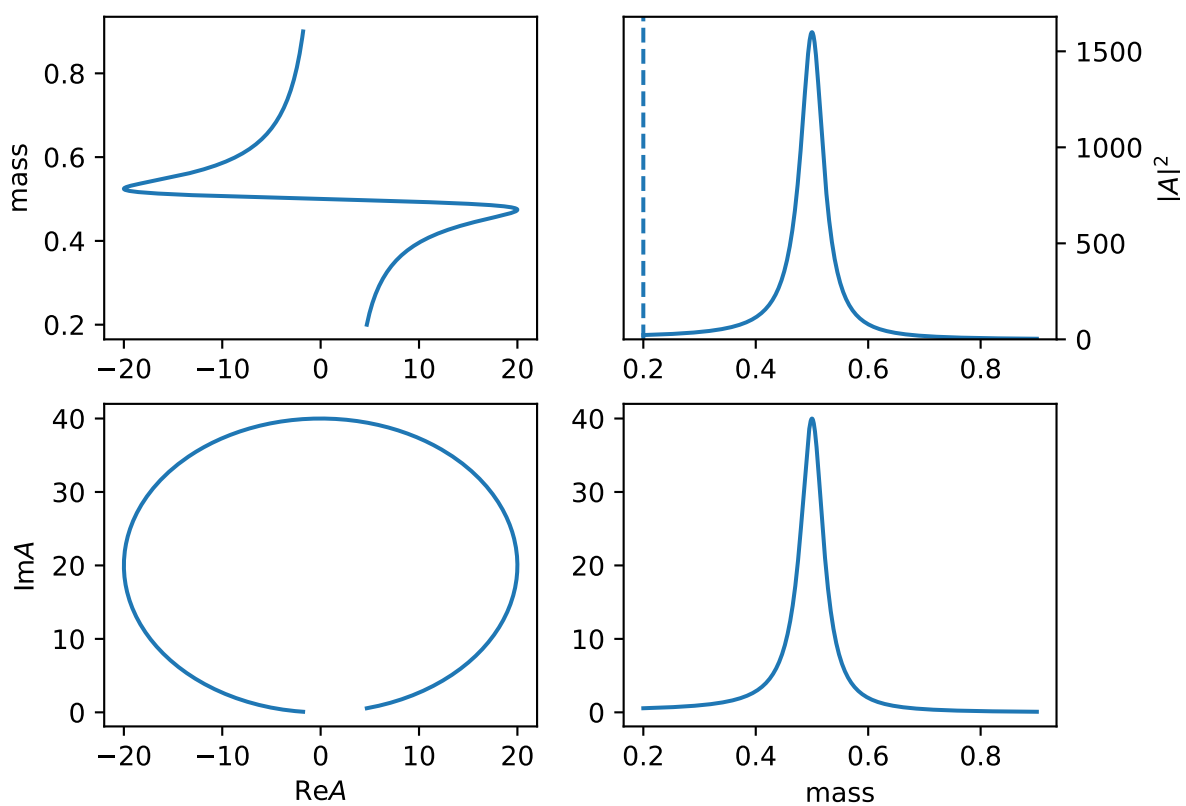
10. "one" (*ParticleOne*)

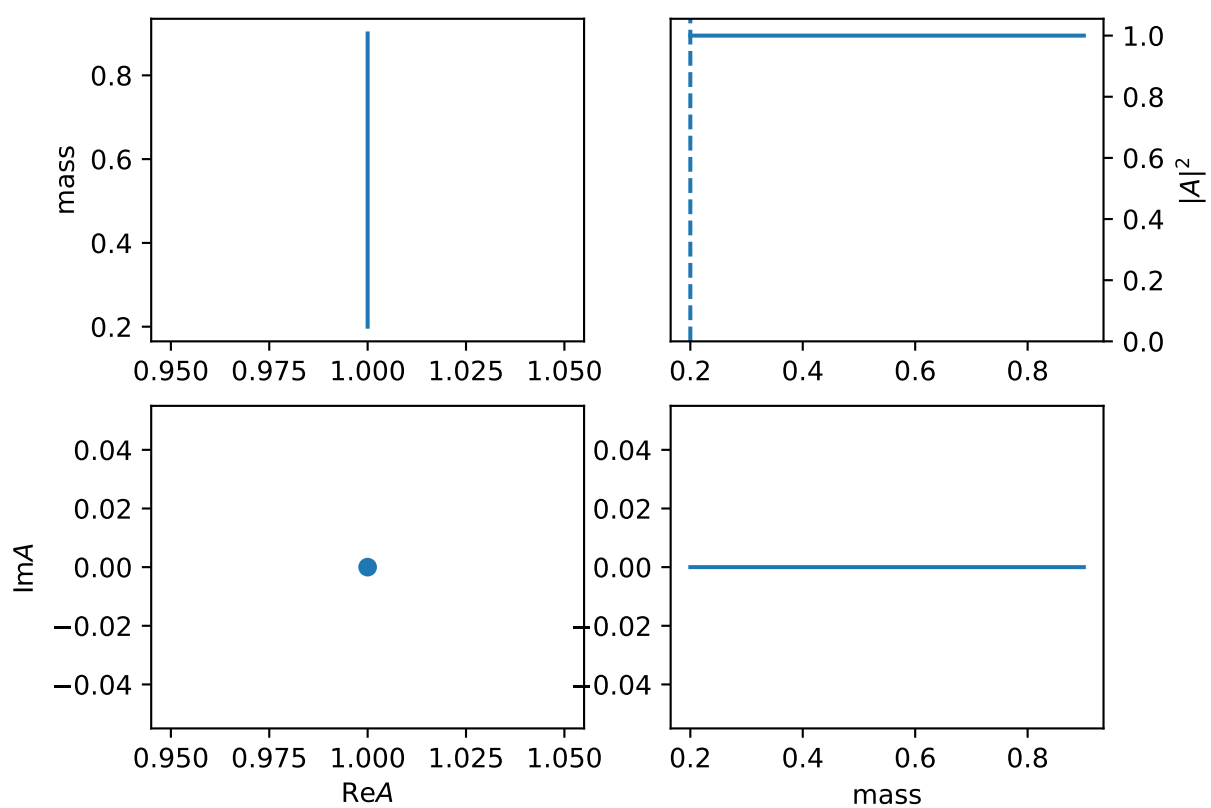
$$R(m) = 1$$

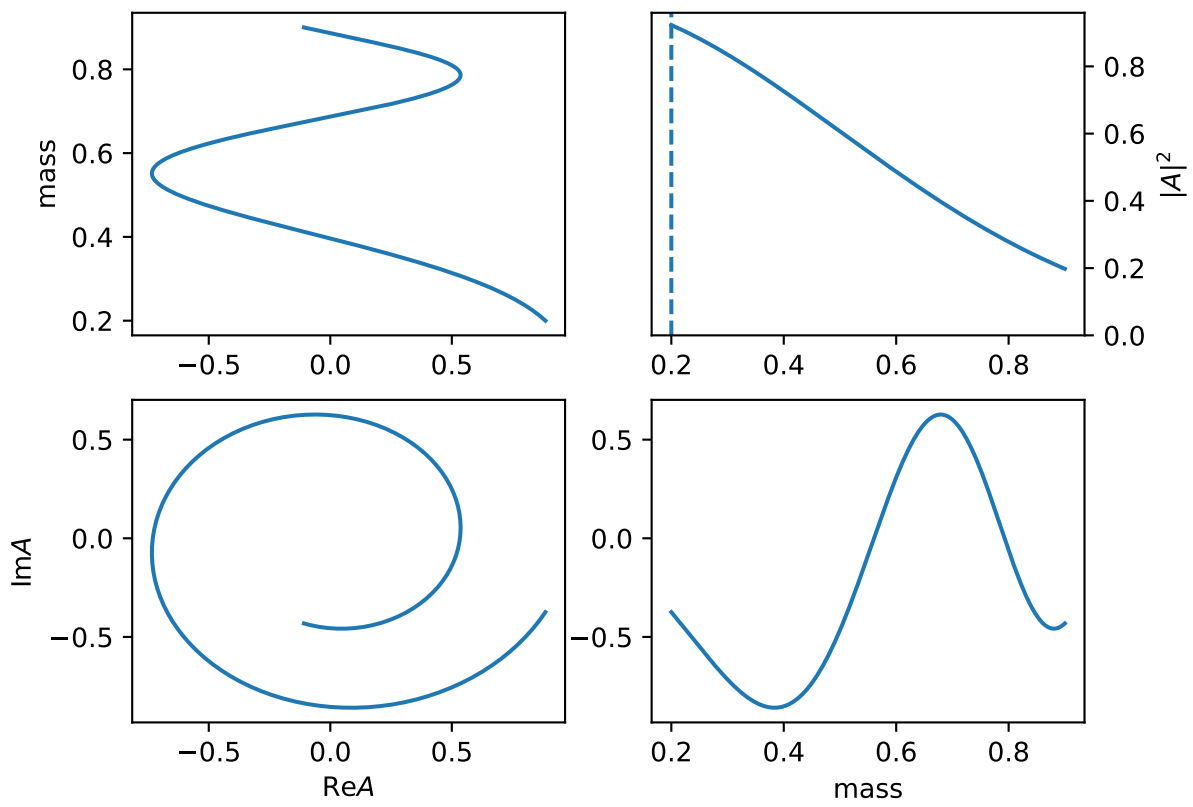
11. "exp" (*ParticleExp*)

$$R(m) = e^{-|a|m}$$

12. "exp_com" (*ParticleExpCom*)







$$R(m) = e^{-(a+ib)m^2}$$

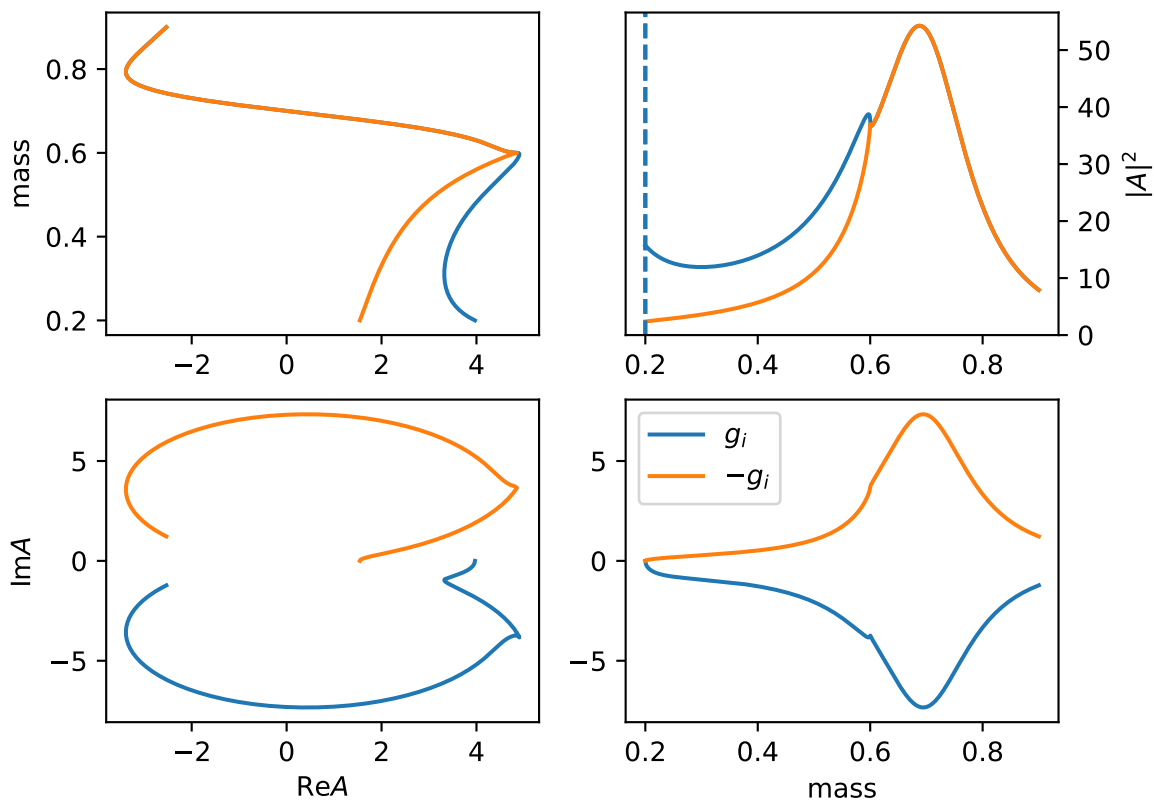
lineshape when $a = 1.0, b = 10$.

13. "Flatte" (*ParticleFlatte*)

Flatte like formula

$$R(m) = \frac{1}{m_0^2 - m^2 + im_0(\sum_i g_i \frac{q_i}{m})}$$

$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$



Required input arguments `mass_list`: `[[m11, m12], [m21, m22]]` for $m_{i,1}, m_{i,2}$.

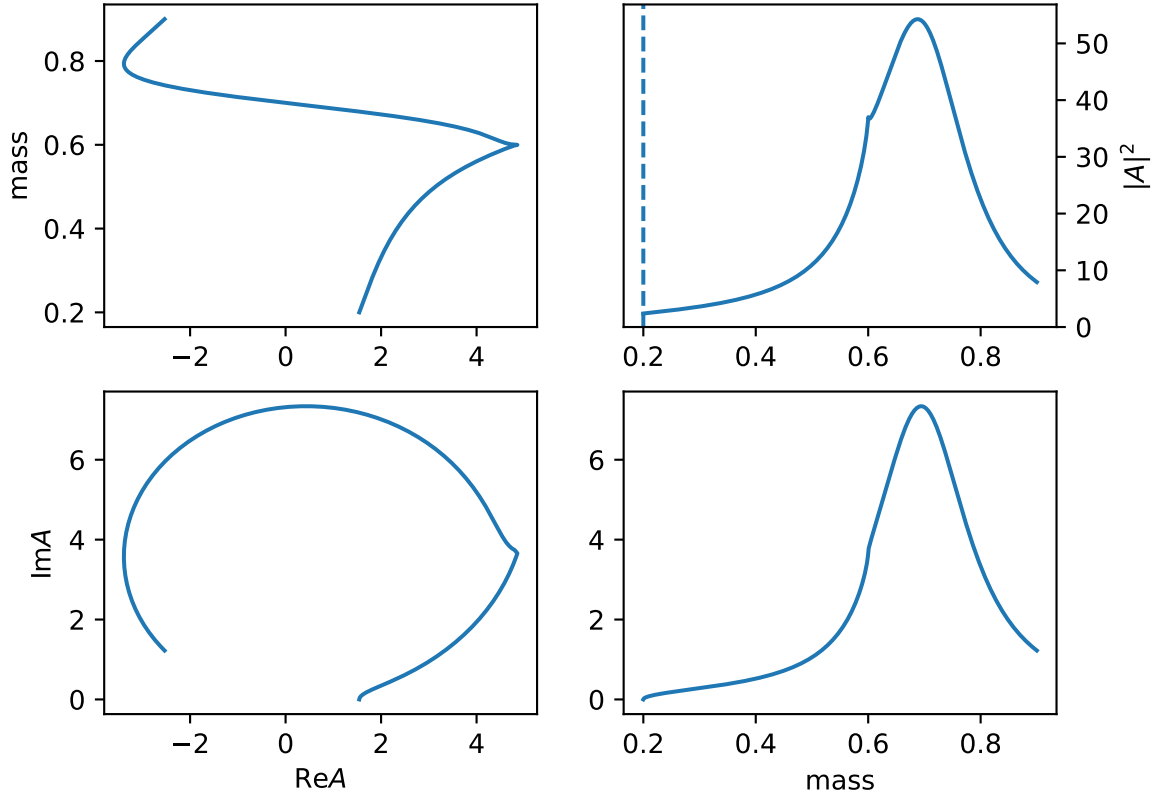
14. "FlatteC" (*ParticleFlatteC*)

Flatte like formula

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0(\sum_i g_i \frac{q_i}{m})}$$

$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$

Required input arguments mass_list: [[m11, m12], [m21, m22]] for $m_{i,1}, m_{i,2}$.



15. "FlatteGen" (*ParticleFlateGen*)

More General Flatte like formula

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0[\sum_i g_i \frac{q_i}{m} \times \frac{m_0}{|q_{i0}|} \times \frac{|q_i|^{2l_i}}{|q_{i0}|^{2l_i}} B_l'^2(|q_i|, |q_{i0}|, d)]}$$

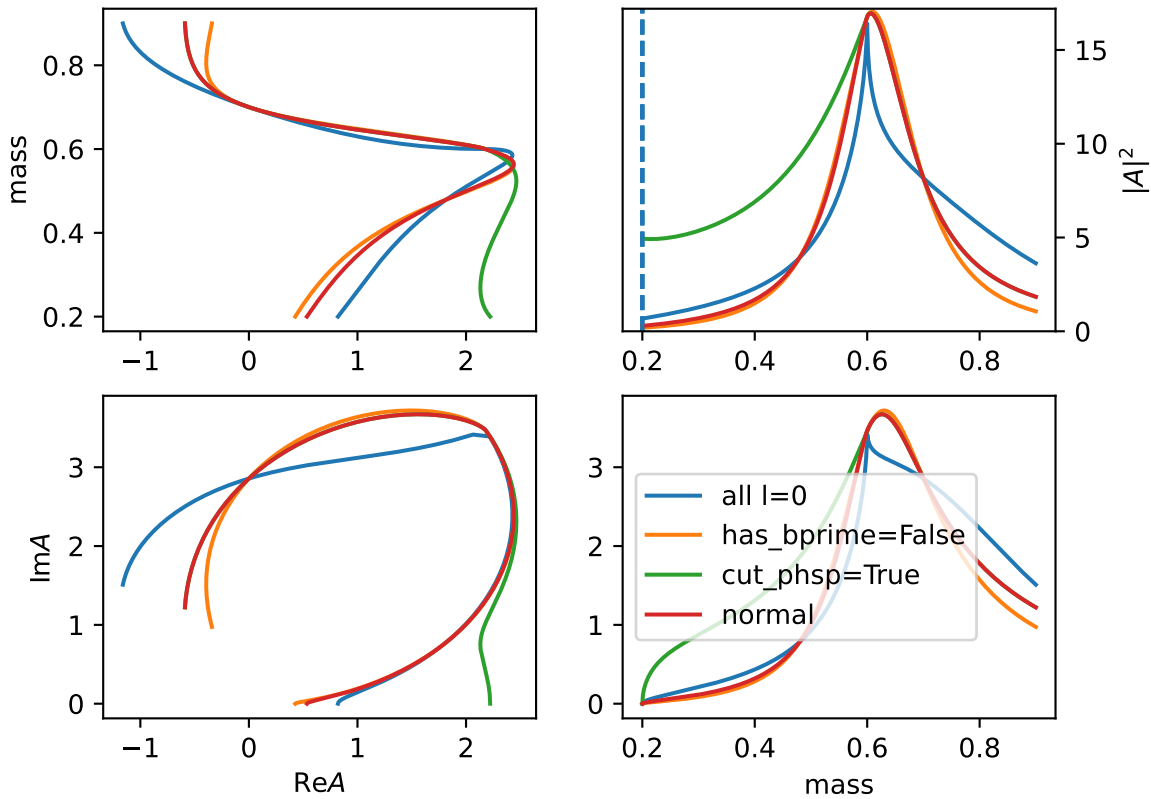
$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$

Required input arguments mass_list: [[m11, m12], [m21, m22]] for $m_{i,1}, m_{i,2}$. And addition arguments l_list: [l1, l2] for l_i

has_bprime=False to remove $B_{l_i}^{\prime 2}(|q_i|, |q_{i0}|, d)$.

cut_phsp=True to set $q_i = 0$ when $(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0$

The plot use parameters $m_0 = 0.7, m_{0,1} = m_{0,2} = 0.1, m_{1,1} = m_{1,2} = 0.3, g_0 = 0.3, g_1 = 0.2, l_0 = 0, l_1 = 1$.



no_m0=True to set $im_0 \Rightarrow i$ in the width part.

no_q0=True to remove $\frac{m_0}{|q_{i0}|}$ and set others $q_{i0} = 1$.

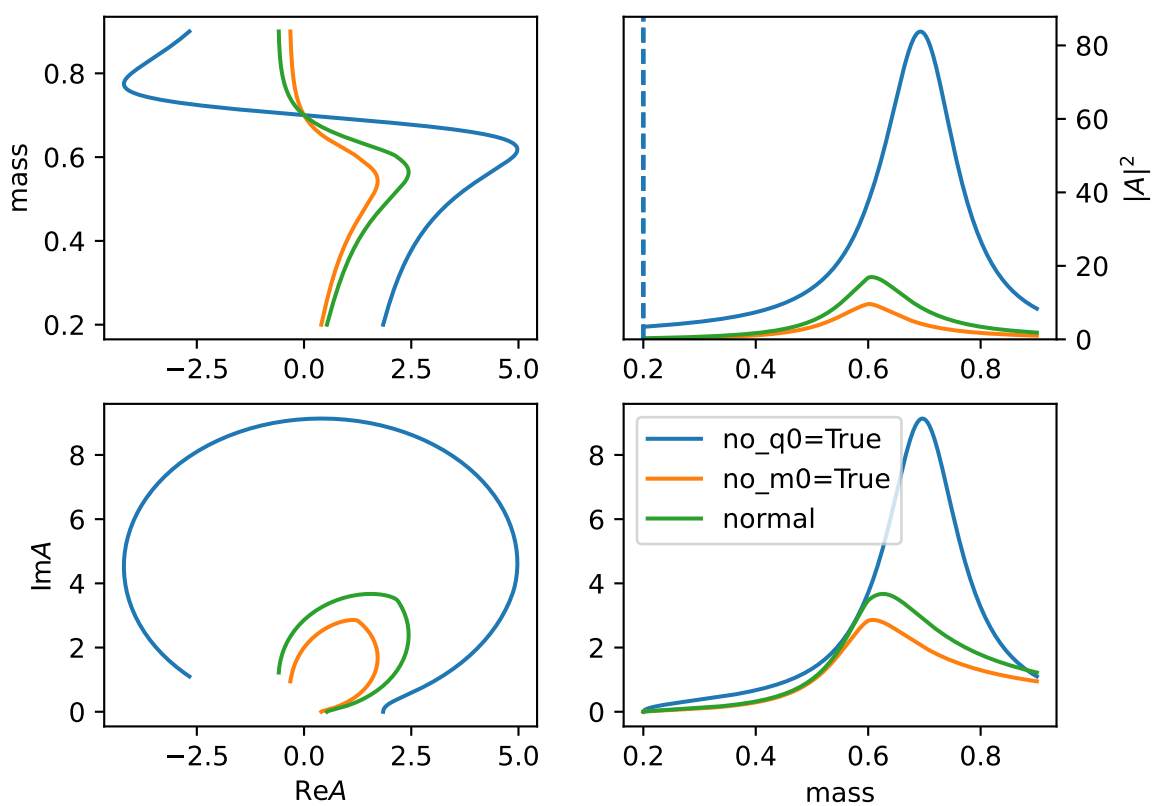
16. "Flatte2" (*ParticleFlatte2*)

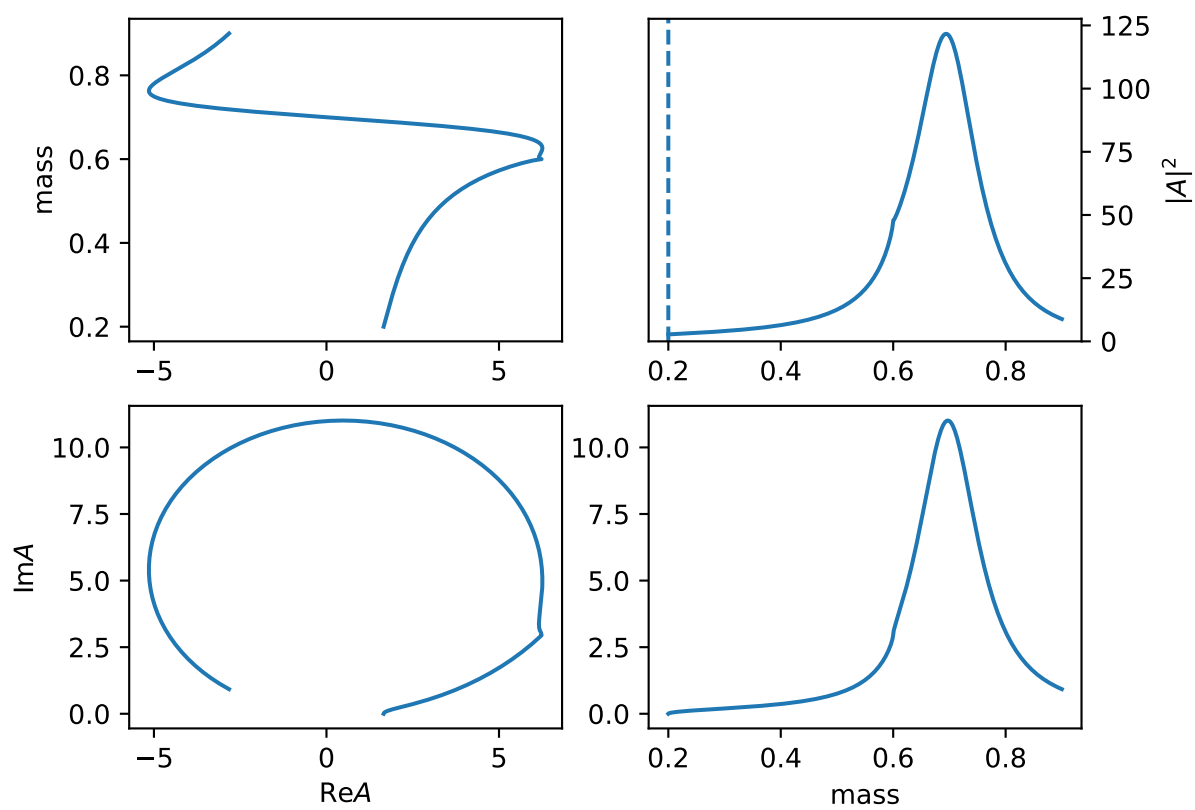
General Flatte like formula.

$$R(m) = \frac{1}{m_0^2 - m^2 - im_0 [\sum_i g_i^2 \frac{q_i}{m} \times \frac{m_0}{|q_{i0}|} \times \frac{|q_i|^{2l_i}}{|q_{i0}|^{2l_i}} B_{l_i}^{\prime 2}(|q_i|, |q_{i0}|, d)]}$$

$$q_i = \begin{cases} \frac{\sqrt{(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) \geq 0 \\ i \frac{\sqrt{|(m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2)|}}{2m} & (m^2 - (m_{i,1} + m_{i,2})^2)(m^2 - (m_{i,1} - m_{i,2})^2) < 0 \end{cases}$$

It has the same options as FlatteGen.





17. "KMatrixSingleChannel" (*KmatrixSingleChannelParticle*)

K matrix model for single channel multi pole.

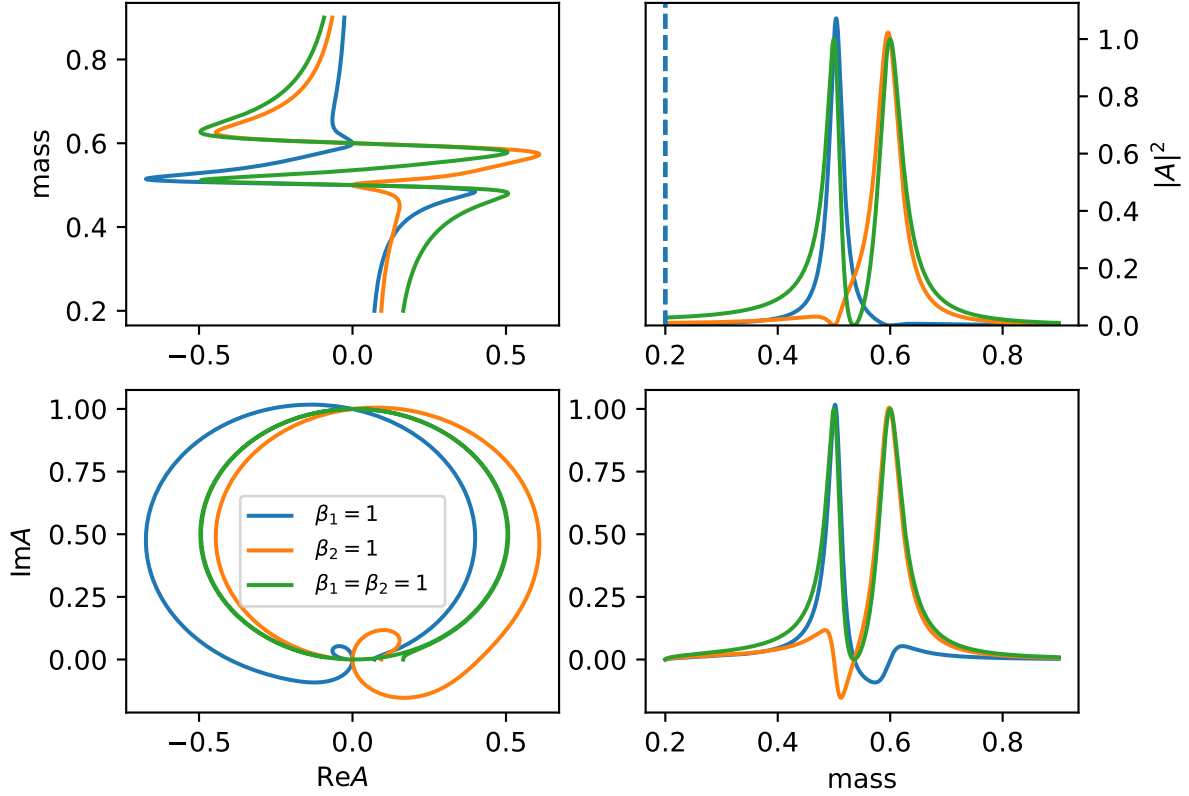
$$K = \sum_i \frac{m_i \Gamma_i(m)}{m_i^2 - m^2}$$

$$P = \sum_i \frac{\beta_i m_0 \Gamma_0}{m_i^2 - m^2}$$

the barrier factor is included in gls

$$R(m) = (1 - iK)^{-1}P$$

requird mass_list: [pole1, pole2] and width_list: [width1, width2].

18. "KMatrixSplitLS" (*KmatrixSplitLSParticle*)

K matrix model for single channel multi pole and the same channel with different (l, s) coupling.

$$K_{a,b} = \sum_i \frac{m_i \sqrt{\Gamma_{a,i}(m) \Gamma_{b,i}(m)}}{m_i^2 - m^2}$$

$$P_b = \sum_i \frac{\beta_i m_0 \Gamma_{b,i0}}{m_i^2 - m^2}$$

the barrier factor is included in gls

$$R(m) = (1 - iK)^{-1}P$$

19. "KmatrixSimple" (*KmatrixSimple*)

simple Kmatrix formula.

K-matrix

$$K_{i,j} = \sum_a \frac{g_{i,a}g_{j,a}}{m_a^2 - m^2 + i\epsilon}$$

P-vector

$$P_i = \sum_a \frac{\beta_a g_{i,a}}{m_a^2 - m^2 + i\epsilon} + f_{bkg,i}$$

total amplitude

$$R(m) = n(1 - Ki\rho n^2)^{-1}P$$

barrief factor

$$n_{ii} = q_i^l B_l'(q_i, 1/d, d)$$

phase space factor

$$\rho_{ii} = q_i/m$$

q_i is 0 when below threshold

20. "BWR_LS" (*ParticleBWRLS*)

Breit Wigner with split ls running width

$$R_i(m) = \frac{g_i}{m_0^2 - m^2 - im_0\Gamma_0 \frac{\rho}{\rho_0} (\sum_i g_i^2)}$$

, $\rho = 2q/m$, the partial width factor is

$$g_i = \gamma_i \frac{q^l}{q_0^l} B_{l_i}'(q, q_0, d)$$

and keep normalize as

$$\sum_i \gamma_i^2 = 1.$$

The normalize is done by $(\cos \theta_0, \sin \theta_0 \cos \theta_1, \dots, \prod_i \sin \theta_i)$

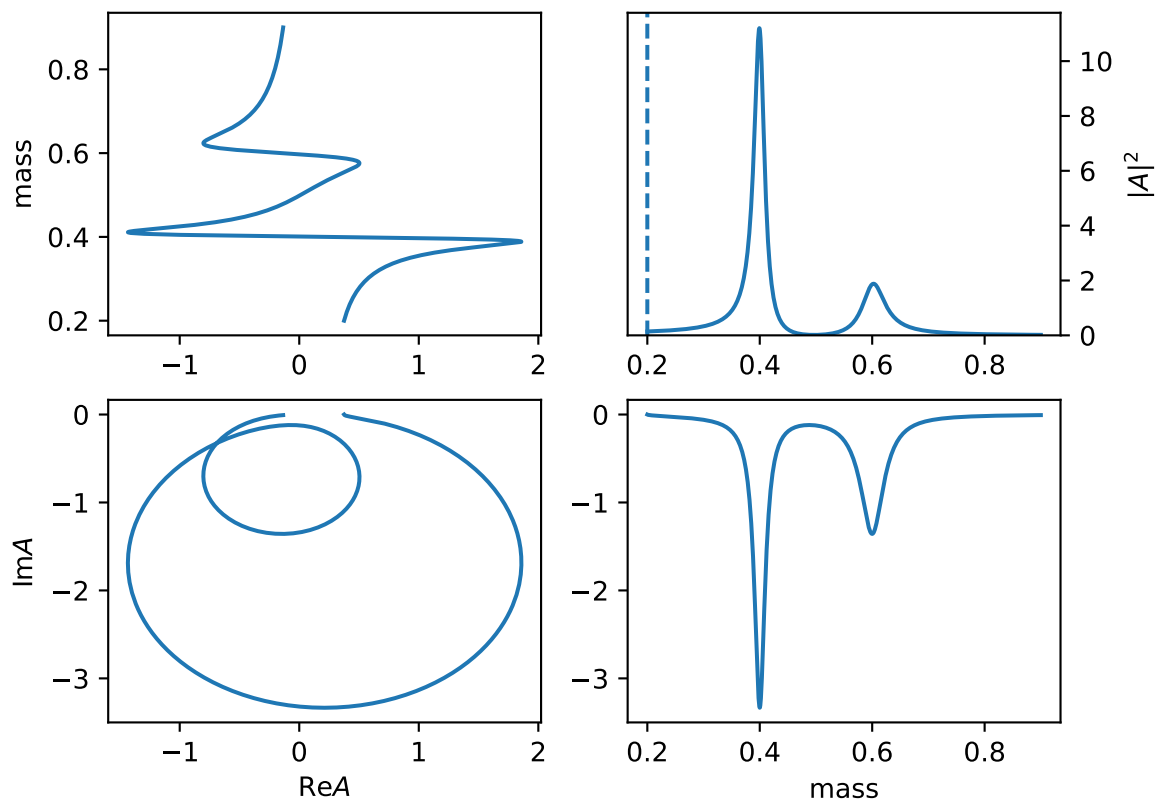
21. "BWR_LS2" (*ParticleBWRLS2*)

Breit Wigner with split ls running width, each one use their own l,

$$R_i(m) = \frac{1}{m_0^2 - m^2 - im_0\Gamma_0 \frac{\rho}{\rho_0} (g_i^2)}$$

, $\rho = 2q/m$, the partial width factor is

$$g_i = \gamma_i \frac{q^l}{q_0^l} B_{l_i}'(q, q_0, d)$$



22. "MultiBWR" (*ParticleMultiBWR*)

Combine Multi BWR into one particle

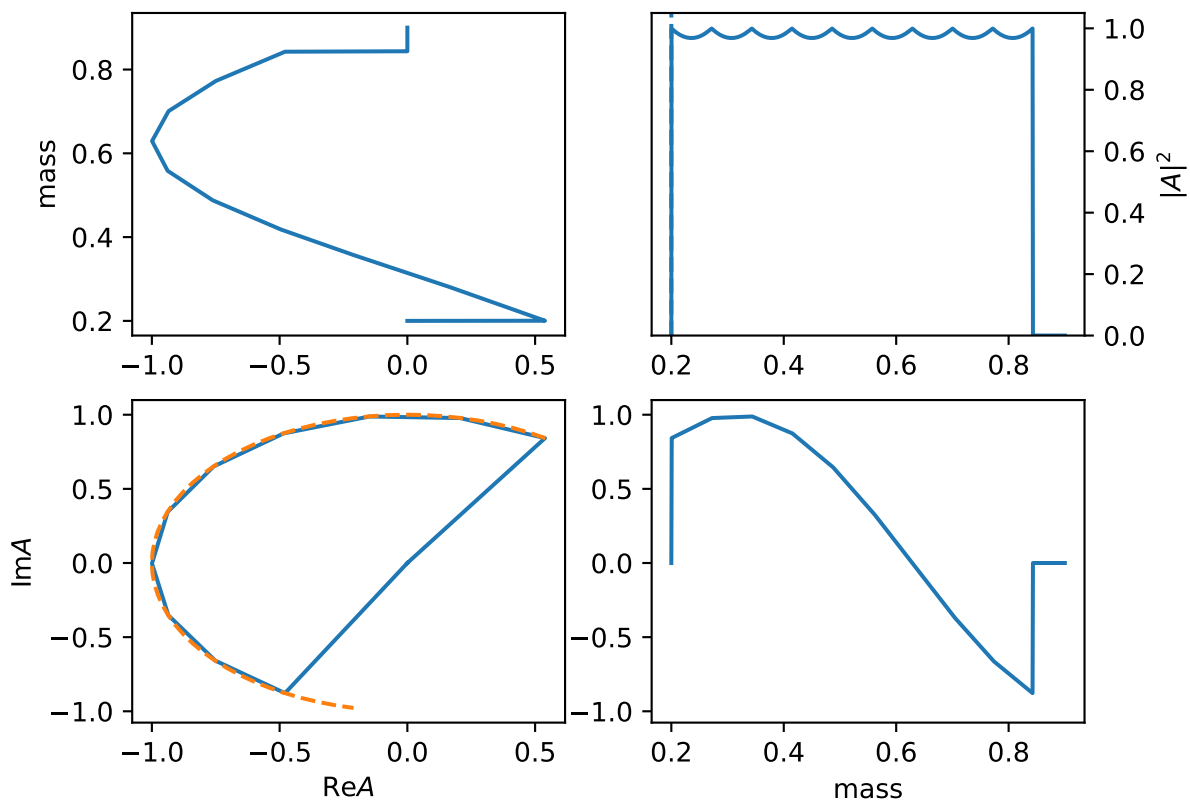
23. "MultiBW" (*ParticleMultiBW*)

Combine Multi BW into one particle

24. "linear_npy" (*InterpLinearNpy*)

Linear interpolation model from a `numpy` file with array of `[mi, re(ai), im(ai)]`. Required file: `path_of_file.npy`, for the path of `numpy` file.

The example is `exp(5 I m)`.

25. "linear_txt" (*InterpLinearTxt*)

Linear interpolation model from a `txt` file with array of `[mi, re(ai), im(ai)]`.

Required file: `path_of_file.txt`, for the path of `txt` file.

The example is `exp(5 I m)`.

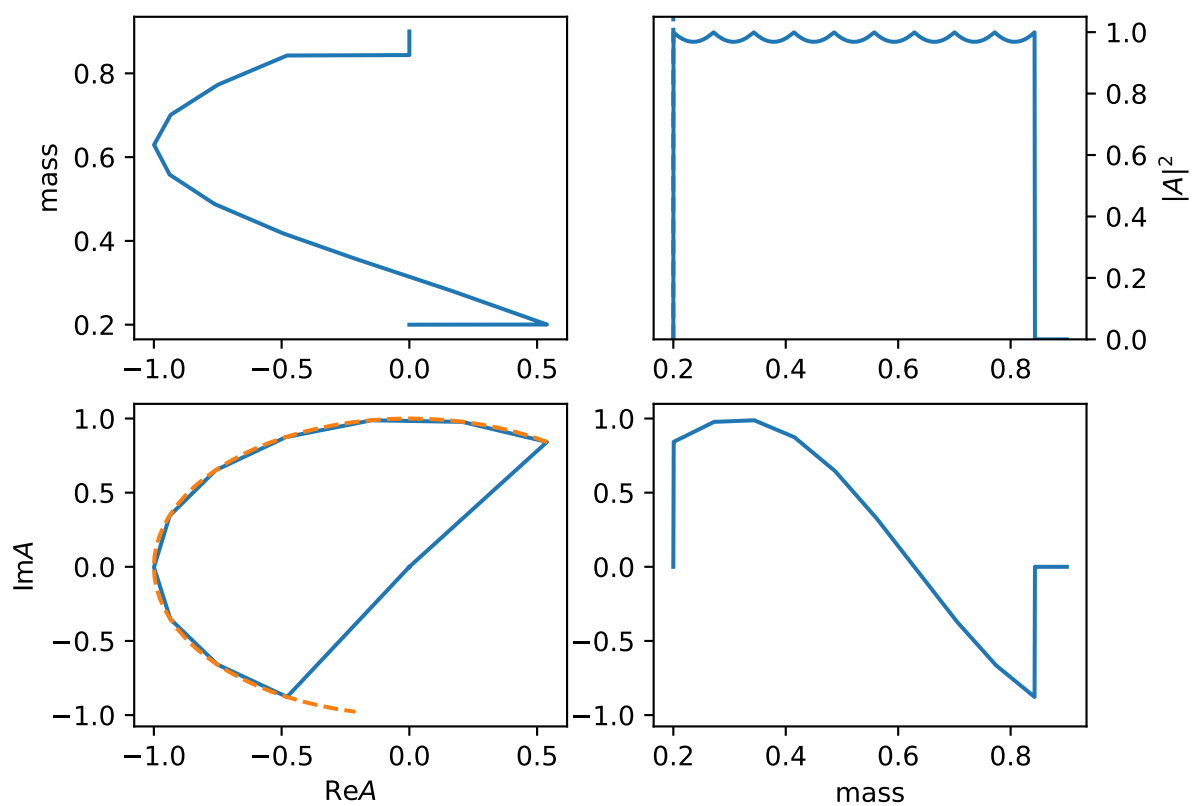
26. "interp" (*Interp*)

linear interpolation for real number

27. "interp_c" (*Interp*)

linear interpolation for complex number

28. "spline_c" (*Interp1DSpline*)



Spline interpolation function for model independent resonance

29. "interp1d3" ([Interp1D3](#))

Piecewise third order interpolation

30. "interp_lagrange" ([Interp1DLang](#))

Lagrange interpolation

31. "interp_hist" ([InterpHist](#))

Interpolation for each bins as constant

32. "hist_idx" ([InterpHistIdx](#))

Interpolation for each bins as constant

use

```
min_m: 0.19
max_m: 0.91
interp_N: 8
with_bound: True
```

for mass range [0.19, 0.91] and 7 bins

The first and last are fixed to zero unless set with_bound: True.

This is an example of $k \exp(ik)$ for point k.

33. "spline_c_idx" ([Interp1DSplineIdx](#))

Spline function in index way.

use

```
min_m: 0.19
max_m: 0.91
interp_N: 8
with_bound: True
```

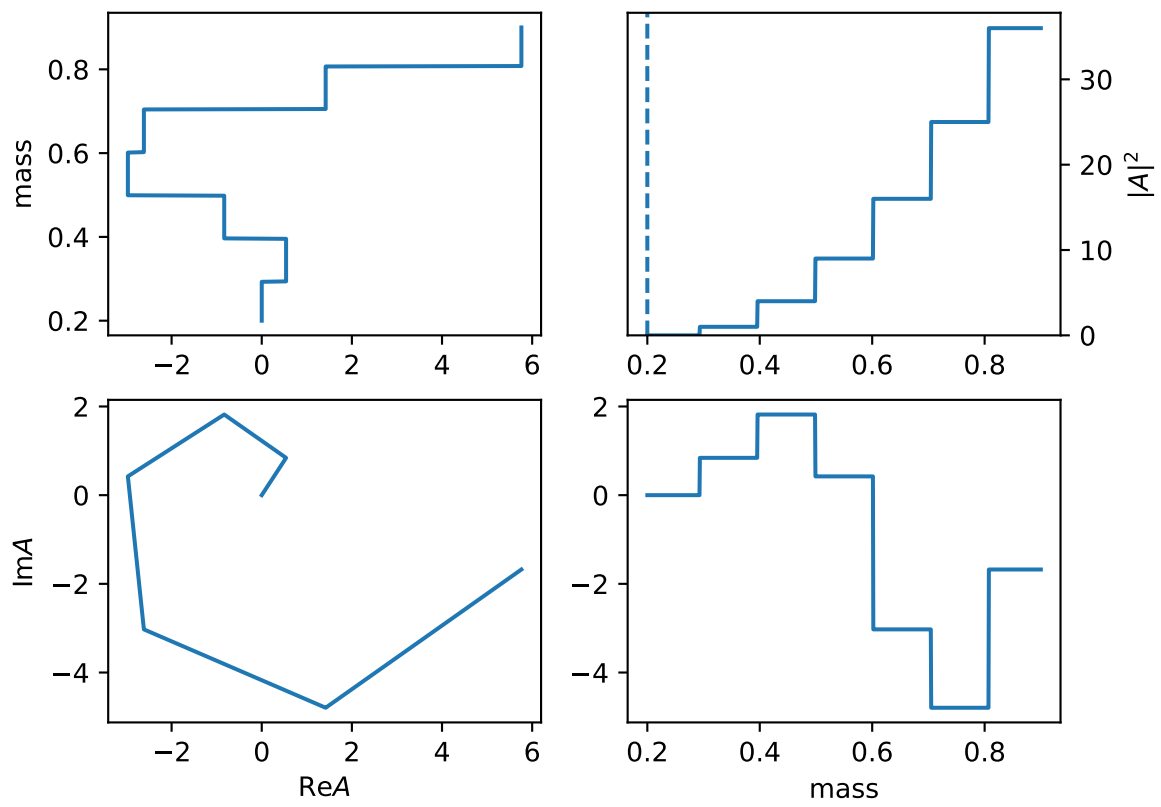
for mass range [0.19, 0.91] and 8 interpolation points

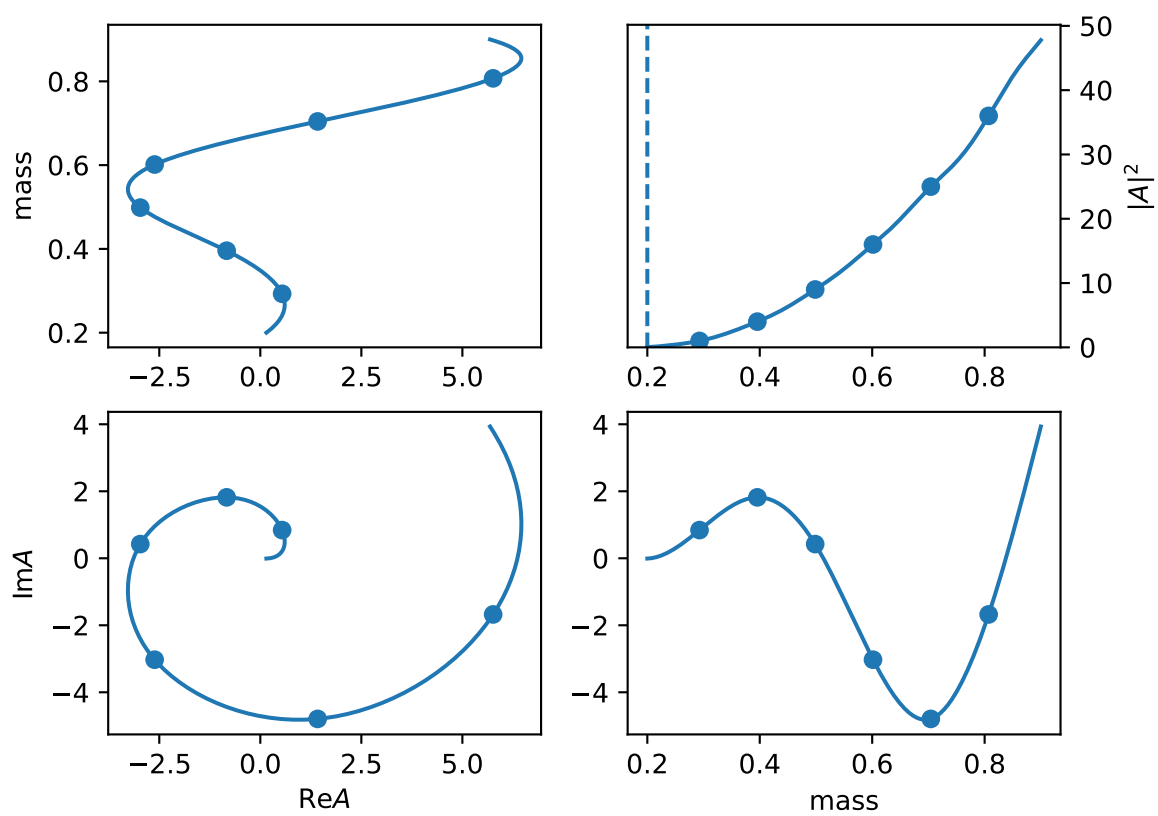
The first and last are fixed to zero unless set with_bound: True.

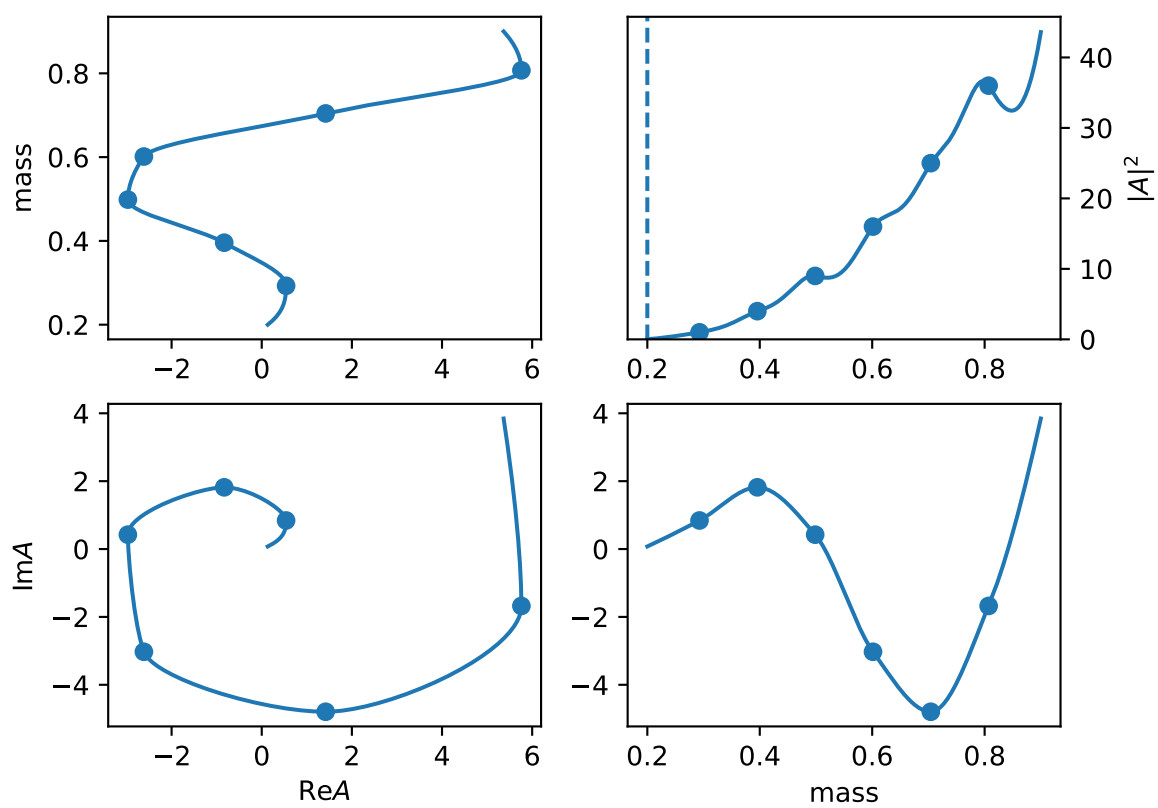
This is an example of $k \exp(ik)$ for point k.

34. "sppchip" ([InterpSPCHIP](#))

Shape-Preserving Piecewise Cubic Hermite Interpolation Polynomial. It is monotonic in each interval.







AVAILABLE DECAY MODEL

12.1 2-body decays

1. "gls-bf", "default" (*HelicityDecay*)

default decay model

The total amplitude is

$$A = H_{\lambda_B, \lambda_C}^{A \rightarrow B+C} D_{\lambda_A, \lambda_B - \lambda_C}^{J_A*}(\varphi, \theta, 0)$$

The helicity coupling is

$$H_{\lambda_B, \lambda_C}^{A \rightarrow B+C} = \sum_{l_s} g_{l_s} \sqrt{\frac{2l+1}{2J_A+1}} \langle l0; s\delta | J_A \delta \rangle \langle J_B \lambda_B; J_C - \lambda_C | s\delta \rangle q^l B_l'(q, q_0, d)$$

The fit parameters is g_{l_s}

There are some options

- (1). `has_bprime=False` will remove the $B_l'(q, q_0, d)$ part.
- (2). `has_barrier_factor=False` will remove the $q^l B_l'(q, q_0, d)$ part.
- (3). `barrier_factor_norm=True` will replace q^l with $(q/q_0)^l$
- (4). `below_threshold=True` will replace the mass used to calculate q_0 with

$$m_0^{eff} = m^{min} + \frac{m^{max} - m^{min}}{2} \left(1 + \tanh \frac{m_0 - \frac{m^{max} + m^{min}}{2}}{m^{max} - m^{min}} \right)$$

(5). `l_list=[l1, l2]` and `ls_list=[[l1, s1], [l2, s2]]` options give the list of all possible LS used in the decay.

(6). `no_q0=True` will set the $q_0 = 1$.

2. "helicity_full" (*HelicityDecayNP*)

Full helicity amplitude

$$A = H_{m_1, m_2} D_{m_0, m_1 - m_2}^{J_0*}(\varphi, \theta, 0)$$

fit parameters is H_{m_1, m_2} .

3. "helicity_parity" (*HelicityDecayP*)

$$H_{-m_1, -m_2} = P_0 P_1 P_2 (-1)^{J_1 + J_2 - J_0} H_{m_1, m_2}$$

4. "gls-cpv" (*HelicityDecayCPV*)

decay model for CPV

TENSORFLOW AND CUDATOOLKIT VERSION

1. Why are there two separate conda requirements file?

- `requirements-min.txt` limits the tensorflow version up to 2.2. Beyond this version, conda will install the wrong dependency versions, in particular cudatoolkit versions and sometimes python3.
- `tensorflow_2_6_requirements.txt` manually selects the correct python and cudatoolkit versions to match the tensorflow-2.6.0 build on conda-forge.

2. Should I use the latest tensorflow version?

- We **highly recommend** Ampere card users (RTX 30 series for example), to install their conda environments with `tensorflow_2_6_requirements.txt` which uses cudatoolkit version **11.2**.

3. Why should Ampere use cudatoolkit version > 11.0?

- To avoid *a few minutes* of overhead due to JIT compilation.
- cudatoolkit version < **11.0** does not have pre-compiled CUDA binaries for Ampere architecture. So older cudatoolkit versions have to JIT compile the PTX code everytime tensorflow uses the GPU hence the overhead.
- See this [explanation](#) about old CUDA versions and JIT compile.

4. Will you update the `tensorflow_2_X_requirements.txt` file regularly to the latest available version on `conda`?

- We do not guarantee any regular updates on `tensorflow_2_X_requirements.txt`.
- We will update this should particular build become unavailable on conda **or** a new release of GPUs require a tensorflow and cudatoolkit update. Please notify us if this is the case.

14.1 1. Precision Loss

message: Desired error **not** necessarily achieved due to precision loss.

Check the jac value,

- 1.1 If all absolute value is small. it is acceptable because of the precision.
- 1.2 If some absolute value is large. It is the bad parameters or problem in models.
- 1.3 Avoid negative weights

14.2 2. NaN value in fit

message: NaN result encountered.

14.2.1 2.1 Check the data.

There is a script (scripts/check_nan.py) to check it.

- 2.1.1 No strange value in data, (nan, infs ...).
- 2.1.2 The data order should be E, p_x, p_y, p_z , E is the first.
- 2.1.3 The mass should be valid, $E^2 - p_x^2 - p_y^2 - p_z^2 > 0$, and for any combinations of final particles, $m_{ab} > m_a + m_b$.
- 2.1.4 Avoid 0 in weights.

14.2.2 2.2 Check the model.

- 2.2.1 The resonance mass should be valid, for example in the mass range (m_1+m_2 , m_0-m_3), out of the threshold required special options.

14.3 3. NaN value when getting params error.

```
numpy.linalg.LinAlgError: Array must not contain infs or NaN.
```

3.1 Similar as sec 2.2.

3.2 Bad fit parameters: too wide width or too narrow width, reach the boundary and so on.

3.3 Bad gradients. No gradients or the gradients is not correct for fit parameters.

14.4 4. Singular Matrix when getting params error

```
numpy.linalg.LinAlgError: Singular matrix
```

4.1 Free parameters are not used in model.

4.2 Used numpy for calculation of variable. The calculation have to be done in get_amp with TensorFlow.

```
...
def init_params(self):
    self.a = self.add_var("a")
def get_amp(self, data, *args, **kwargs):
    # avoid use numpy for variable as
    a = np.sin(self.a())
    # use tensorflow instead
    a = tf.sin(self.a())
```

14.5 5. Out of memory (OOM)

14.5.1 5.1 GPU

```
tensorflow.python.framework.errors_impl.ResourceExhaustedError: OOM when_
→allocating tensor with shape ... device:GPU:0 by allocator GPU_0_bfc [Op:...]
```

5.1.1 Reduce batch size at `config.fit(batch=65000)` and `config.get_params_error(batch=13000)` in `fit.py`.

5.1.2 Use option for large data size, such as lazy call

```
# config.yml
data:
    lazy_call: True
```

5.1.3 Try to use small data sample, or simple cases (less final particles).

5.1.4 Some special model required large memory (such as interpolation model), try other model.

14.5.2 5.2 CPU

killed

5.2.1 Try to allocate more memory. There should be some options for job.

5.2.2 Similar as sec 5.1

14.6 6. Bad config.yml

14.6.1 6.1 yaml parse error

`yaml.parser.ParserError: while parsing ..`

Check the yaml file (see <https://yaml.org>): the indent, speical chars , :}], unicode and so on.

14.6.2 6.2 Decay chain

`AssertionError: not only one top particle`

The decay chain should be complete. All the item in decay should decay from initial to finals.

14.6.3 6.3 Decay chain 2

`RuntimeError: not decay chain aviable, check you config.yml`

6.3.1 Similar as sec 6.2.

6.3.2 Check the information in *remove decay chain*, see the reson why those decays are not aviable.

ls not aviable means no possible LS combination allowed. Check the spin and parity. If allow parity voilate, add `p_break: True` to decay.

FOR STARTERS

Take the decay in `config_sample.yml` for example, we will generate a toy, and then use it to conduct an analysis.

First, we use the class `ConfigLoader` to load the configuration file, and thus building the amplitude expression.

```
from tf_pwa.config_loader import ConfigLoader
config = ConfigLoader("config_sample.yml")
amp = config.get_amplitude()
```

We can also use a json file to set the parameters in the amplitude formula `config.set_params("parameters.json")`. Otherwise, the parameters are set randomly.

Next, we can use functions in `tf_pwa.applications` to directly generate data samples. We need to provide the masses of A and B , C , D to generate the PhaseSpace MC, and then use it to generate the toy data.

```
from tf_pwa.applications import gen_mc, gen_data
PHSP = gen_mc(mother=4.6, daughters=[2.00698, 2.01028, 0.13957], number=100000, outfile=
    ↪ "PHSP.dat")
data = gen_data(amp, Ndata=5000, mcfile="PHSP.dat", genfile="data.dat")
```

Now that we have updated `data.dat` and `PHSP.dat`, we'd better load the configuration file again, and then fit the data.

```
config = ConfigLoader("config_sample.yml")
fit_result = config.fit()
```

Fitting is the major part of an analysis, and it could also be the most time-consuming part. For this example (the complexity of the amplitude expression matters a lot), the time for fitting is about xxx (running on xxxGPU). Then we can step further to complete the analysis, like calculating the fit fractions.

```
errors = config.get_params_error(fit_result)
fit_result.save_as("final_parameters.json")
fit_frac, err_frac = config.cal_fitfractions()
```

We can use `error_print` in `tf_pwa.utils` to print the fitting parameters as well as the fit fractions.

```
from tf_pwa.utils import error_print
print("##### fitting parameters:")
for key, value in config.get_params().items():
    print(key, error_print(value, errors.get(key, None)))
print("##### fit fractions:")
for i in fit_frac:
    print(i, " : " + error_print(fit_frac[i], err_frac.get(i, None)))
```

If the plotting options are also provided in the `config_sample.yml`, we can also plot the distributions of variables indicated in the configuration file.

```
config.plot_partial_wave(fit_result, prefix="figure/")
```

The figures will be saved under path `figure`. Here are the three invariant mass pairs for example.

(three pictures here)

We can do a lot more using `tf_pwa`. For more examples, please see path `tutorials`.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

t

tf_pwa, 41
tf_pwa.adaptive_bins, 117
tf_pwa.amp, 41
tf_pwa.amp.amp, 44
tf_pwa.amp.base, 45
tf_pwa.amp.core, 54
tf_pwa.amp.cov_ten, 63
tf_pwa.amp.flatte, 65
tf_pwa.amp.interpolation, 70
tf_pwa.amp.Kmatrix, 41
tf_pwa.amp.kmatrix_simple, 77
tf_pwa.amp.preprocess, 77
tf_pwa.amp.split_ls, 78
tf_pwa.angle, 118
tf_pwa.app, 81
tf_pwa.app.fit, 81
tf_pwa.applications, 121
tf_pwa.breit_wigner, 125
tf_pwa.cal_angle, 127
tf_pwa.cg, 132
tf_pwa.config, 132
tf_pwa.config_loader, 81
tf_pwa.config_loader.base_config, 81
tf_pwa.config_loader.config_loader, 81
tf_pwa.config_loader.data, 85
tf_pwa.config_loader.data_root_lhcb, 86
tf_pwa.config_loader.decay_config, 87
tf_pwa.config_loader.extra, 87
tf_pwa.config_loader.multi_config, 87
tf_pwa.config_loader.particle_function, 88
tf_pwa.config_loader.plot, 88
tf_pwa.config_loader.plotter, 90
tf_pwa.config_loader.sample, 93
tf_pwa.cov_ten_ir, 133
tf_pwa.data, 133
tf_pwa.data_trans, 94
tf_pwa.data_trans.dalitz, 94
tf_pwa.data_trans.helicity_angle, 94
tf_pwa.dec_parser, 137
tf_pwa.dfun, 138
tf_pwa.einsum, 140
tf_pwa.err_num, 140
tf_pwa.experimental, 95
tf_pwa.experimental.build_amp, 95
tf_pwa.experimental.extra_amp, 95
tf_pwa.experimental.extra_data, 95
tf_pwa.experimental.extra_function, 96
tf_pwa.experimental.factor_system, 96
tf_pwa.experimental.opt_int, 97
tf_pwa.experimental.wrap_function, 97
tf_pwa.expermental, 141
tf_pwa.fit, 141
tf_pwa.fit_improve, 142
tf_pwa.fitfractions, 144
tf_pwa.formula, 145
tf_pwa.function, 145
tf_pwa.generator, 97
tf_pwa.generator.breit_wigner, 97
tf_pwa.generator.generator, 98
tf_pwa.generator.interp_nd, 98
tf_pwa.generator.linear_interpolation, 99
tf_pwa.generator.plane_2d, 99
tf_pwa.generator.square_dalitz_plot, 100
tf_pwa.gpu_info, 145
tf_pwa.histogram, 145
tf_pwa.main, 147
tf_pwa.model, 102
tf_pwa.model.cfit, 102
tf_pwa.model.custom, 104
tf_pwa.model.model, 106
tf_pwa.model.opt_int, 115
tf_pwa.params_trans, 147
tf_pwa.particle, 147
tf_pwa.phasespace, 152
tf_pwa.root_io, 154
tf_pwa.significance, 154
tf_pwa.tensorflow_wrapper, 154
tf_pwa.transform, 155
tf_pwa.utils, 155
tf_pwa.variable, 157
tf_pwa.version, 165
tf_pwa.vis, 165
tf_pwa.weight_smear, 166

A

- AbsPDF (class in *tf_pwa.amp.amp*), 44
- adaptive_shape() (in module *tf_pwa.adaptive_bins*), 118
- AdaptiveBound (class in *tf_pwa.adaptive_bins*), 117
- add() (Count method), 97
- add() (Seq method), 142
- add_algin() (HelicityDecay method), 57
- add_cartesiancp_var() (VarsManager method), 161
- add_complex_var() (VarsManager method), 161
- add_constraints() (ConfigLoader method), 81
- add_creator() (BaseParticle method), 148
- add_decay() (BaseParticle method), 148
- add_decay_constraints() (ConfigLoader method), 81
- add_fix_var_constraints() (ConfigLoader method), 81
- add_free_var_constraints() (ConfigLoader method), 81
- add_from_trans_constraints() (ConfigLoader method), 81
- add_gauss_constr_constraints() (ConfigLoader method), 81
- add_gen() (GenTest method), 98
- add_mass() (in module *tf_pwa.cal_angle*), 129
- add_particle_constraints() (ConfigLoader method), 81
- add_pre_trans_constraints() (ConfigLoader method), 81
- add_real_var() (VarsManager method), 161
- add_ref_amp() (Plotter method), 90
- add_relative_momentum() (in module *tf_pwa.cal_angle*), 129
- add_used_chains() (DecayGroup method), 55
- add_var() (AmpBase method), 54
- add_var_equal_constraints() (ConfigLoader method), 81
- add_var_range_constraints() (ConfigLoader method), 81
- add_weight() (in module *tf_pwa.cal_angle*), 129
- AfterGenerator (class in *tf_pwa.config_loader.sample*), 93
- aligned_angle_ref_rule1() (in module *tf_pwa.cal_angle*), 129
- aligned_angle_ref_rule2() (in module *tf_pwa.cal_angle*), 129
- AlignmentAngle (class in *tf_pwa.angle*), 118
- amp0ls() (in module *tf_pwa.cov_ten_ir*), 133
- amp2s() (ParticleFunction method), 88
- amp_index() (AmpDecay method), 54
- amp_index() (DecayChain method), 55
- amp_index() (DecayGroup method), 55
- amp_matrix_as_dict() (in module *tf_pwa.experimental.build_amp*), 95
- amp_shape() (AmpBase method), 54
- amp_shape() (AmpDecay method), 54
- amp_shape() (DecayChain method), 55
- amp_shape() (Particle method), 58
- AmpBase (class in *tf_pwa.amp.core*), 54
- AmpDecay (class in *tf_pwa.amp.core*), 54
- AmpDecayChain (class in *tf_pwa.amp.core*), 54
- AmplitudeModel (class in *tf_pwa.amp.amp*), 44
- angle_from() (Vector3 method), 120
- angle_px_px() (AlignmentAngle static method), 118
- angle_zx_z_getx() (EulerAngle static method), 118
- angle_zx_zx() (EulerAngle static method), 119
- angle_zx_zzz_getx() (EulerAngle static method), 119
- AngSam3Decay (class in *tf_pwa.amp.core*), 54
- append_int() (FitFractions method), 144
- apply() (NumberError method), 140
- arg_max() (Seq method), 142
- ARGenerator (class in *tf_pwa.generator.generator*), 98
- array_split() (in module *tf_pwa.utils*), 155
- as_config() (BaseDecay method), 147
- as_config() (BaseParticle method), 148
- as_config() (DecayGroup method), 150
- as_dataset() (LazyCall method), 134
- as_dataset() (LazyFile method), 135
- attach_fix_params_error() (ConfigLoader method), 81
- AttrDict (class in *tf_pwa.utils*), 155

B

- barrier_factor() (Decay method), 148

- barrier_factor() (in module *tf_pwa.amp.kmatrix_simple*), 77
 barrier_factor() (in module *tf_pwa.breit_wigner*), 126
 barrier_factor2() (in module *tf_pwa.breit_wigner*), 126
 base_bound() (*AdaptiveBound* static method), 117
 BaseAmplitudeModel (class in *tf_pwa.amp.amp*), 44
 BaseConfig (class in *tf_pwa.config_loader.base_config*), 81
 BaseCustomModel (class in *tf_pwa.model.custom*), 104
 BaseDecay (class in *tf_pwa.particle*), 147
 BaseGenerator (class in *tf_pwa.generator.generator*), 98
 BaseModel (class in *tf_pwa.model.model*), 106
 BaseParticle (class in *tf_pwa.particle*), 147
 BasePreProcessor (class in *tf_pwa.amp.preprocess*), 77
 BaseTransform (class in *tf_pwa.transform*), 155
 batch() (*LazyCall* method), 134
 batch_call() (in module *tf_pwa.data*), 135
 batch_call_numpy() (in module *tf_pwa.data*), 135
 batch_sum() (in module *tf_pwa.data*), 135
 batch_sum_var() (*ConfigLoader* method), 82
 batch_sum_var() (*VarsManager* method), 162
 beta() (*LorentzVector* method), 119
 bin_center (*Hist1D* property), 145
 bin_width (*Hist1D* property), 145
 binning_shape_function() (in module *tf_pwa.adaptive_bins*), 118
 Bl() (in module *tf_pwa.amp.Kmatrix*), 41
 boost() (*LorentzVector* method), 119
 boost_matrix() (*LorentzVector* method), 119
 boost_vector() (*LorentzVector* method), 119
 Boost_z() (*SU2M* static method), 120
 Boost_z_from_p() (*SU2M* static method), 120
 Bound (class in *tf_pwa.variable*), 157
 Bprime() (in module *tf_pwa.breit_wigner*), 126
 Bprime_num() (in module *tf_pwa.breit_wigner*), 126
 Bprime_polynomial() (in module *tf_pwa.breit_wigner*), 126
 Bprime_polynomial() (in module *tf_pwa.formula*), 145
 Bprime_q2() (in module *tf_pwa.breit_wigner*), 126
 build_amp2s() (in module *tf_pwa.experimental.build_amp*), 95
 build_amp_matrix() (in module *tf_pwa.experimental.build_amp*), 95
 build_angle_amp_matrix() (in module *tf_pwa.experimental.build_amp*), 95
 build_barrier_factor() (*KmatrixSimple* method), 77
 build_cached() (*CachedAnglePreProcessor* method), 77
 build_cached() (*CachedShapePreProcessor* method), 78
 build_cached_int() (*ModelCachedInt* method), 116
 build_coeffs() (*InterpND* method), 98
 build_coeffs() (*InterpNDHist* method), 98
 build_coupling_einsum() (*CovTenDecayChain* method), 63
 build_data() (*HelicityAngle* method), 94
 build_decay_einsum() (*CovTenDecayChain* method), 63
 build_einsum() (*CovTenDecayChain* method), 63
 build_int_matrix() (in module *tf_pwa.experimental.opt_int*), 97
 build_int_matrix_batch() (in module *tf_pwa.experimental.opt_int*), 97
 build_k_matrix() (*KmatrixSimple* method), 77
 build_l_einsum() (*CovTenDecayChain* method), 63
 build_ls2hel_eq() (*HelicityDecay* method), 57
 build_matrix() (in module *tf_pwa.config_loader.data_root_lhcb*), 86
 build_p_vector() (*KmatrixSimple* method), 77
 build_params_matrix() (in module *tf_pwa.experimental.opt_int*), 97
 build_params_vector() (in module *tf_pwa.experimental.build_amp*), 95
 build_params_vector() (in module *tf_pwa.experimental.opt_int*), 97
 build_phsp_chain() (in module *tf_pwa.config_loader.sample*), 93
 build_phsp_chain_sorted() (in module *tf_pwa.config_loader.sample*), 93
 build_read_var_function() (in module *tf_pwa.config_loader.plot*), 88
 build_s_einsum() (*CovTenDecayChain* method), 63
 build_simple_data() (*HelicityDecay* method), 57
 build_sum_amplitude() (in module *tf_pwa.experimental.build_amp*), 95
 build_sum_amplitude() (in module *tf_pwa.experimental.opt_int*), 97
 build_sum_angle_amplitude() (in module *tf_pwa.experimental.build_amp*), 95
 build_wave_function() (*CovTenDecayChain* method), 63
 BW() (in module *tf_pwa.breit_wigner*), 125
 BW_dom() (in module *tf_pwa.formula*), 145
 BWGenerator (class in *tf_pwa.generator.breit_wigner*), 97
 BWR() (in module *tf_pwa.breit_wigner*), 125
 BWR2() (in module *tf_pwa.breit_wigner*), 125
 BWR_coupling_dom() (in module *tf_pwa.formula*), 145
 BWR_dom() (in module *tf_pwa.formula*), 145
 BWR_LS_dom() (in module *tf_pwa.formula*), 145
 BWR_normal() (in module *tf_pwa.breit_wigner*), 125
- ## C
- cache_data() (*BaseAmplitudeModel* method), 44

`cached_amp()` (in `tf_pwa.experimental.build_amp`), 95
`cached_amp2s()` (in `tf_pwa.experimental.build_amp`), 95
`cached_available()` (*AbsPDF method*), 44
`cached_available()` (*BaseAmplitudeModel method*), 44
`Cached_FG` (class in `tf_pwa.fit_improve`), 142
`cached_int_mc()` (in `tf_pwa.experimental.opt_int`), 97
`CachedAmpAmplitudeModel` (class in `tf_pwa.amp.amp`), 44
`CachedAmpPreProcessor` (class in `tf_pwa.amp.preprocess`), 77
`CachedAnglePreProcessor` (class in `tf_pwa.amp.preprocess`), 77
`CachedShapeAmplitudeModel` (class in `tf_pwa.amp.amp`), 44
`CachedShapePreProcessor` (class in `tf_pwa.amp.preprocess`), 78
`cal_1d_shape()` (*TriangleGenerator method*), 99
`cal_angle()` (*HelicityAngle method*), 94
`cal_angle()` (in module `tf_pwa.cal_angle`), 129
`cal_angle()` (*P4DirectlyAmplitudeModel method*), 45
`cal_angle()` (*SimpleData method*), 85
`cal_angle_from_momentum()` (in module `tf_pwa.cal_angle`), 129
`cal_angle_from_momentum_base()` (in module `tf_pwa.cal_angle`), 130
`cal_angle_from_momentum_id_swap()` (in module `tf_pwa.cal_angle`), 130
`cal_angle_from_momentum_single()` (in module `tf_pwa.cal_angle`), 130
`cal_angle_from_particle()` (in module `tf_pwa.cal_angle`), 130
`cal_bins_numbers()` (*ConfigLoader method*), 82
`cal_bins_numbers()` (in module `tf_pwa.config_loader.extra`), 87
`cal_chain_boost()` (in module `tf_pwa.cal_angle`), 130
`cal_chi2()` (*ConfigLoader method*), 82
`cal_chi2()` (in module `tf_pwa.adaptive_bins`), 118
`cal_chi2()` (in module `tf_pwa.config_loader.extra`), 87
`cal_coeff_left()` (*TriangleGenerator method*), 99
`cal_coeff_right()` (*TriangleGenerator method*), 99
`cal_coeffs()` (*LinearInterp method*), 99
`cal_coeffs()` (*TriangleGenerator method*), 99
`cal_err()` (in module `tf_pwa.err_num`), 140
`cal_fitfractions()` (*ConfigLoader method*), 82
`cal_fitfractions()` (in module `tf_pwa.fitfractions`), 144
`cal_fitfractions_no_grad()` (in module `tf_pwa.fitfractions`), 144
`cal_helicity_angle()` (in module `tf_pwa.cal_angle`), 130
`cal_hesse_correct()` (in module `tf_pwa.applications`), 121
`cal_hesse_error()` (in module `tf_pwa.applications`), 121
`cal_max_weight()` (*AfterGenerator method*), 93
`cal_max_weight()` (*ChainGenerator method*), 152
`cal_max_weight()` (*PhaseSpaceGenerator method*), 152
`cal_monentum()` (in module `tf_pwa.amp.flatte`), 69
`cal_monentum_sympy()` (in module `tf_pwa.amp.flatte`), 70
`cal_signal_yields()` (*ConfigLoader method*), 82
`cal_significance()` (in module `tf_pwa.applications`), 122
`cal_single_boost()` (in module `tf_pwa.cal_angle`), 130
`cal_st_xy()` (*TriangleGenerator method*), 99
`cal_xy_st()` (*TriangleGenerator method*), 99
`CalAngleData` (class in `tf_pwa.cal_angle`), 128
`call()` (*BaseTransform method*), 155
`call()` (*EinSum method*), 64
`call()` (*EinSumCall method*), 64
`call()` (*EvalBoost method*), 64
`call()` (*LinearTrans method*), 155
`cauchy()` (in module `tf_pwa.histogram`), 146
`cg_coef()` (in module `tf_pwa.cg`), 132
`cg_in_amp0ls()` (in module `tf_pwa.cov_ten_ir`), 133
`chain_decay()` (*BaseParticle method*), 148
`ChainGenerator` (class in `tf_pwa.phasespace`), 152
`chains_particle()` (*BaseAmplitudeModel method*), 44
`chains_particle()` (*DecayGroup method*), 55
`check_nan()` (in module `tf_pwa.data`), 135
`check_positive_definite()` (in module `tf_pwa.utils`), 155
`check_valid_jp()` (*ConfigLoader method*), 82
`check_valid_jp()` (*HelicityDecay method*), 57
`chi2()` (*Hist1D method*), 145
`clip_log()` (in module `tf_pwa.model.model`), 113
`CombineFCN` (class in `tf_pwa.model.model`), 107
`combineVM()` (in module `tf_pwa.variable`), 165
`compare_result()` (in module `tf_pwa.applications`), 122
`ConfigLoader` (class in `tf_pwa.config_loader.config_loader`), 81
`ConfigManager` (class in `tf_pwa.config`), 132
`ConstrainModel` (class in `tf_pwa.model.model`), 108
`copy()` (*LazyCall method*), 134
`corr_coef_matrix()` (in module `tf_pwa.applications`), 122
`cos_theta()` (*Vector3 method*), 121
`Count` (class in `tf_pwa.experimental.wrap_function`), 97
`CovTenDecayChain` (class in `tf_pwa.amp.cov_ten`), 63
`CovTenDecayNew` (class in `tf_pwa.amp.cov_ten`), 63
`CovTenDecaySimple` (class in `tf_pwa.amp.cov_ten`), 63

- `cp_charge_group()` (in module `tf_pwa.particle`), 151
`cp_swap_p()` (in module `tf_pwa.cal_angle`), 130
`cplx_cpvar()` (Variable method), 159
`cplx_var()` (Variable method), 159
`create_amplitude()` (in module `tf_pwa.amp.amp`), 45
`create_cal_calangle()` (in module `tf_pwa.config_loader.sample`), 93
`create_chain_property()` (in module `tf_pwa.config_loader.plot`), 88
`create_complex_root_sympy_tfop()` (in module `tf_pwa.formula`), 145
`create_config()` (in module `tf_pwa.config`), 132
`create_data()` (RootData method), 86
`create_dir()` (in module `tf_pwa.utils`), 155
`create_epsilon()` (in module `tf_pwa.amp.cov_ten`), 64
`create_new()` (LazyCall method), 134
`create_new()` (LazyFile method), 135
`create_plot_var_dic()` (in module `tf_pwa.config_loader.plot`), 88
`create_preprocessor()` (in module `tf_pwa.amp.preprocess`), 78
`create_rotate_p()` (in module `tf_pwa.data_trans.helicity_angle`), 95
`create_rotate_p_decay()` (in module `tf_pwa.data_trans.helicity_angle`), 95
`create_sppchip_matrix()` (in module `tf_pwa.amp.interpolation`), 76
`create_test_config()` (in module `tf_pwa.utils`), 155
`create_trans()` (in module `tf_pwa.transform`), 155
`cross()` (Vector3 method), 121
`cross_combine()` (in module `tf_pwa.particle`), 151
`cross_unit()` (Vector3 method), 121
`custom_cond()` (in module `tf_pwa.config_loader.data_root_lhcb`), 86
`cut_data()` (in module `tf_pwa.config_loader.data_root_lhcb`), 86
- ## D
- `D_matrix_conj()` (in module `tf_pwa.dfun`), 138
`Dalitz` (class in `tf_pwa.data_trans.dalitz`), 94
`data_cut()` (in module `tf_pwa.data`), 135
`data_device()` (in module `tf_pwa.amp.core`), 60
`data_generator()` (in module `tf_pwa.data`), 135
`data_index()` (in module `tf_pwa.data`), 135
`data_map()` (in module `tf_pwa.data`), 135
`data_mask()` (in module `tf_pwa.data`), 135
`data_merge()` (in module `tf_pwa.data`), 136
`data_replace()` (in module `tf_pwa.data`), 136
`data_shape()` (in module `tf_pwa.data`), 136
`data_split()` (in module `tf_pwa.data`), 136
`data_strip()` (in module `tf_pwa.data`), 136
`data_struct()` (in module `tf_pwa.data`), 136
`data_to_numpy()` (in module `tf_pwa.data`), 136
`data_to_tensor()` (in module `tf_pwa.data`), 136
`DataType` (BaseGenerator attribute), 98
`DataType` (BWGenerator attribute), 97
`DataType` (LinearInterp attribute), 99
`DataType` (LinearInterpImportance attribute), 99
`Decay` (class in `tf_pwa.particle`), 148
`decay_chain_cut()` (DecayConfig method), 87
`decay_chain_cut_list` (DecayConfig attribute), 87
`decay_cut()` (DecayConfig method), 87
`decay_cut_list` (DecayConfig attribute), 87
`decay_cut_ls()` (in module `tf_pwa.config_loader.decay_config`), 87
`decay_cut_mass()` (in module `tf_pwa.config_loader.decay_config`), 87
`decay_item()` (DecayConfig static method), 87
`DecayChain` (class in `tf_pwa.amp.core`), 54
`DecayChain` (class in `tf_pwa.particle`), 149
`DecayConfig` (class in `tf_pwa.config_loader.decay_config`), 87
`DecayGroup` (class in `tf_pwa.amp.core`), 55
`DecayGroup` (class in `tf_pwa.particle`), 150
`deep_iter()` (in module `tf_pwa.utils`), 155
`deep_ordered_iter()` (in module `tf_pwa.utils`), 155
`deep_ordered_range()` (in module `tf_pwa.utils`), 155
`deep_stack()` (in module `tf_pwa.variable`), 165
`default_color_generator()` (in module `tf_pwa.config_loader.plot`), 88
`delta_D_index()` (in module `tf_pwa.dfun`), 139
`delta_D_trans()` (in module `tf_pwa.dfun`), 139
`delta_idx_in_amp0ls()` (in module `tf_pwa.cov_ten_ir`), 133
`density()` (ParticleFunction method), 88
`depth_first()` (DecayChain method), 149
`dFun()` (in module `tf_pwa.breit_wigner`), 127
`Dfun_delta()` (in module `tf_pwa.dfun`), 138
`Dfun_delta_v2()` (in module `tf_pwa.dfun`), 139
`dh_dsFun()` (in module `tf_pwa.breit_wigner`), 127
`dirichlet_smear()` (in module `tf_pwa.weight_smear`), 166
`disable_allow_cc()` (DecayConfig method), 87
`do_command()` (in module `tf_pwa.dec_parser`), 137
`do_spline_hmatrix()` (in module `tf_pwa.amp.interpolation`), 76
`dom_fun()` (ParticleMultiBW method), 79
`dom_fun()` (ParticleMultiBWR method), 79
`dot()` (in module `tf_pwa.amp.cov_ten`), 64
`Dot()` (LorentzVector method), 119
`dot()` (Vector3 method), 121
`dot_chain()` (DotGenerator static method), 165
`dot_default_edge` (DotGenerator attribute), 165
`dot_default_node` (DotGenerator attribute), 165
`dot_head` (DotGenerator attribute), 165
`dot_label_edge` (DotGenerator attribute), 165
`dot_ranksame` (DotGenerator attribute), 165
`dot_tail` (DotGenerator attribute), 165

DotGenerator (class in *tf_pwa.vis*), 165
 draw() (*Hist1D* method), 145
 draw_bar() (*Hist1D* method), 145
 draw_decay_struct() (in module *tf_pwa.vis*), 165
 draw_error() (*Hist1D* method), 145
 draw_hist() (*Hist1D* method), 145
 draw_kde() (*Hist1D* method), 146
 draw_kde() (*WeightedData* method), 146
 draw_line() (*Hist1D* method), 146
 draw_pull() (*Hist1D* method), 146

E

EinSum (class in *tf_pwa.amp.cov_ten*), 64
 Einsum (class in *tf_pwa.einsum*), 140
 einsum() (in module *tf_pwa.einsum*), 140
 EinSumCall (class in *tf_pwa.amp.cov_ten*), 64
 epanechnikov() (in module *tf_pwa.histogram*), 146
 erfc_inverse() (in module *tf_pwa.significance*), 154
 error (NumberError property), 140
 error_print() (in module *tf_pwa.utils*), 155
 error_trans() (*VarsManager* method), 162
 EulerAngle (class in *tf_pwa.angle*), 118
 eval() (*LazyCall* method), 135
 eval() (*LazyFile* method), 135
 eval_amplitude() (*ConfigLoader* method), 82
 eval_integral() (in module *tf_pwa.fitfractions*), 144
 eval_nll_part() (*BaseCustomModel* method), 104
 eval_nll_part() (*SimpleCFitModel* method), 105
 eval_nll_part() (*SimpleChi2Model* method), 105
 eval_nll_part() (*SimpleClipNllModel* method), 105
 eval_nll_part() (*SimpleNllFracModel* method), 105
 eval_nll_part() (*SimpleNllModel* method), 105
 eval_normal_factors() (*BaseCustomModel* method), 104
 eval_normal_factors() (*SimpleCFitModel* method), 105
 eval_normal_factors() (*SimpleNllFracModel* method), 105
 eval_normal_factors() (*SimpleNllModel* method), 105
 eval_phsp_factor() (*HelicityAngle* method), 94
 EvalBoost (class in *tf_pwa.amp.cov_ten*), 64
 EvalLazy (class in *tf_pwa.data*), 134
 EvalP (class in *tf_pwa.amp.cov_ten*), 64
 EvalT (class in *tf_pwa.amp.cov_ten*), 64
 EvalT2 (class in *tf_pwa.amp.cov_ten*), 64
 except_result() (in module *tf_pwa.fit*), 141
 exp() (*NumberError* method), 140
 exp_i() (in module *tf_pwa.dfun*), 139
 export_legend() (in module *tf_pwa.config_loader.plot*), 88
 extra_function() (in module *tf_pwa.experimental.extra_function*), 96

F

f_bg() (in module *tf_pwa.model.cfit*), 104
 f_eff() (in module *tf_pwa.model.cfit*), 104
 F_Sigma() (in module *tf_pwa.cov_ten_ir*), 133
 factor_gamma() (*ParticleBWRLS* method), 78
 factor_iter_names() (*HelicityDecay* method), 57
 factor_iteration() (*BaseAmplitudeModel* method), 44
 factor_iteration() (*DecayChain* method), 55
 factor_iteration() (*DecayGroup* method), 55
 factor_names() (*Variable* method), 159
 FactorAmplitudeModel (class in *tf_pwa.amp.amp*), 45
 Fb() (in module *tf_pwa.amp.Kmatrix*), 41
 FCN (class in *tf_pwa.model.model*), 109
 find_variable() (*HelicityAngle* method), 94
 fit() (*ConfigLoader* method), 82
 fit() (in module *tf_pwa.app.fit*), 81
 fit() (in module *tf_pwa.applications*), 122
 fit() (*MultiConfig* method), 87
 fit_fractions() (in module *tf_pwa.applications*), 123
 fit_minuit() (in module *tf_pwa.fit*), 141
 fit_minuit_v1() (in module *tf_pwa.fit*), 141
 fit_minuit_v2() (in module *tf_pwa.fit*), 141
 fit_multinest() (in module *tf_pwa.fit*), 141
 fit_newton_cg() (in module *tf_pwa.fit*), 141
 fit_normal() (in module *tf_pwa.utils*), 156
 fit_root_fitter() (in module *tf_pwa.fit*), 141
 fit_scipy() (in module *tf_pwa.fit*), 141
 FitFractions (class in *tf_pwa.fitfractions*), 144
 fitNtimes() (*ConfigLoader* method), 82
 FitResult (class in *tf_pwa.fit*), 141
 fix_unused_h() (*HelicityDecayNP* method), 46
 fixed() (*Variable* method), 159
 FL() (in module *tf_pwa.cov_ten_ir*), 133
 flatten_all() (in module *tf_pwa.experimental.factor_system*), 96
 flatten_dict_data() (in module *tf_pwa.data*), 136
 flatten_dict_data() (in module *tf_pwa.utils*), 156
 flatten_mass() (*PhaseSpaceGenerator* method), 152
 flatten_np_data() (in module *tf_pwa.utils*), 156
 FloatParams (class in *tf_pwa.amp.core*), 56
 fmin_bfgs_f() (in module *tf_pwa.fit_improve*), 142
 force_int() (in module *tf_pwa.cov_ten_ir*), 133
 force_pos_def() (in module *tf_pwa.applications*), 123
 force_pos_def_minuit2() (in module *tf_pwa.applications*), 123
 forzen_style() (*Plotter* method), 90
 Frame (class in *tf_pwa.config_loader.plotter*), 90
 free_for_extended() (*ConfigLoader* method), 82
 freed() (*Variable* method), 159
 from_call() (*SumVar* method), 158
 from_call_with_hess() (*SumVar* method), 158
 from_p4() (*LorentzVector* static method), 119
 from_particles() (*DecayChain* static method), 149

from_sorted_table() (*DecayChain* static method), 149

FS() (in module *tf_pwa.cov_ten_ir*), 133

fsFun() (in module *tf_pwa.breit_wigner*), 127

fun() (*Cached_FG* method), 142

G

Gamma() (in module *tf_pwa.breit_wigner*), 126

gamma() (*LorentzVector* method), 119

Gamma2() (in module *tf_pwa.breit_wigner*), 126

gamma_smeared() (in module *tf_pwa.weight_smeared*), 166

gauss() (in module *tf_pwa.histogram*), 146

GaussianConstr (class in *tf_pwa.model.model*), 110

gen_data() (in module *tf_pwa.applications*), 124

gen_mc() (in module *tf_pwa.applications*), 124

gen_random_charge() (in module *tf_pwa.config_loader.sample*), 93

generate() (*AfterGenerator* method), 93

generate() (*ARGenerator* method), 98

generate() (*BaseGenerator* method), 98

generate() (*BWGenerator* method), 97

generate() (*ChainGenerator* method), 152

generate() (*GenTest* method), 98

generate() (*InterpND* method), 98

generate() (*InterpNDHist* method), 98

generate() (*LinearInterp* method), 99

generate() (*LinearInterpImportance* method), 99

generate() (*PhaseSpaceGenerator* method), 152

generate() (*SDPGenerator* method), 100

generate() (*TriangleGenerator* method), 100

generate() (*UniformGenerator* method), 153

generate_mass() (*PhaseSpaceGenerator* method), 153

generate_momentum() (*PhaseSpaceGenerator* method), 153

generate_new_style() (*StyleSet* method), 92

generate_p() (*Dalitz* method), 94

generate_p() (*HelicityAngle1* method), 94

generate_p() (in module *tf_pwa.data_trans.dalitz*), 94

generate_p() (in module *tf_pwa.data_trans.helicity_angle*), 95

generate_p2() (*HelicityAngle1* method), 94

generate_p_mass() (*HelicityAngle* method), 94

generate_p_mass() (*HelicityAngle1* method), 95

generate_params() (*Decay* method), 148

generate_phasespace() (*DecayGroup* method), 55

generate_phspace() (*ConfigLoader* method), 82

generate_phspace() (in module *tf_pwa.config_loader.sample*), 93

generate_phspace() (in module *tf_pwa.phasespace*), 153

generate_phspace_p() (*ConfigLoader* method), 82

generate_phspace_p() (in module *tf_pwa.config_loader.sample*), 93

generate_SDP() (*ConfigLoader* method), 82

generate_SDP() (in module *tf_pwa.config_loader.sample*), 93

generate_SDP() (in module *tf_pwa.generator.square_dalitz_plot*), 100

generate_SDP_p() (*ConfigLoader* method), 82

generate_SDP_p() (in module *tf_pwa.config_loader.sample*), 93

generate_st() (*TriangleGenerator* method), 100

generate_toy() (*ConfigLoader* method), 82

generate_toy() (in module *tf_pwa.config_loader.sample*), 93

generate_toy2() (*ConfigLoader* method), 82

generate_toy2() (in module *tf_pwa.config_loader.sample*), 93

generate_toy_o() (*ConfigLoader* method), 83

generate_toy_o() (in module *tf_pwa.config_loader.sample*), 93

generate_toy_p() (*ConfigLoader* method), 83

generate_toy_p() (in module *tf_pwa.config_loader.sample*), 93

GenTest (class in *tf_pwa.generator.generator*), 98

get() (*IndexMap* method), 64

get() (*LazyCall* method), 135

get() (*LineStyleSet* method), 88

get() (*StyleSet* method), 92

get() (*VarsManager* method), 162

get_all_amp() (*CovTenDecaySimple* method), 64

get_all_chain() (in module *tf_pwa.experimental.factor_system*), 96

get_all_data() (*ConfigLoader* method), 83

get_all_data() (*MultiConfig* method), 87

get_all_data() (*SimpleData* method), 85

get_all_dic() (*VarsManager* method), 162

get_all_factor() (*DecayChain* method), 55

get_all_frame() (*ConfigLoader* method), 83

get_all_frame() (in module *tf_pwa.config_loader.plotter*), 92

get_all_hist() (*Plotter* method), 91

get_all_histogram() (*PlotAllData* method), 90

get_all_mass() (*HelicityAngle* method), 94

get_all_partial_amp() (in module *tf_pwa.experimental.factor_system*), 96

get_all_particles() (*DecayChain* method), 149

get_all_plotdatas() (*ConfigLoader* method), 83

get_all_plotdatas() (in module *tf_pwa.config_loader.plotter*), 92

get_all_val() (*VarsManager* method), 162

get_amp() (*AngSam3Decay* method), 54

get_amp() (*CovTenDecayChain* method), 63

get_amp() (*CovTenDecayNew* method), 63

get_amp() (*CovTenDecaySimple* method), 64

get_amp() (*DecayChain* method), 55

get_amp() (*DecayGroup* method), 55

- `get_amp()` (*HelicityDecay* method), 57
`get_amp()` (*InterpolationParticle* method), 76
`get_amp()` (*KmatrixSingleChannelParticle* method), 41
`get_amp()` (*KmatrixSplitLSParticle* method), 43
`get_amp()` (*Particle* method), 60
`get_amp()` (*ParticleBW* method), 47
`get_amp()` (*ParticleBWR2* method), 48
`get_amp()` (*ParticleBWR_normal* method), 49
`get_amp()` (*ParticleBWRBelowThreshold* method), 48
`get_amp()` (*ParticleBWRCoupling* method), 49
`get_amp()` (*ParticleExp* method), 50
`get_amp()` (*ParticleExpCom* method), 50
`get_amp()` (*ParticleFlateGen* method), 66
`get_amp()` (*ParticleFlatte* method), 68
`get_amp()` (*ParticleGS* method), 50
`get_amp()` (*ParticleKmatrix* method), 52
`get_amp()` (*ParticleLass* method), 52
`get_amp()` (*ParticleLS* method), 79
`get_amp()` (*ParticleOne* method), 52
`get_amp()` (*ParticleX* method), 60
`get_amp()` (*SimpleResonances* method), 60
`get_amp2()` (*DecayGroup* method), 55
`get_amp3()` (*DecayGroup* method), 55
`get_amp_list()` (*FactorAmplitudeModel* method), 45
`get_amp_list_part()` (*FactorAmplitudeModel* method), 45
`get_amp_particle()` (*DecayChain* method), 55
`get_amp_total()` (*DecayChain* method), 55
`get_amplitude()` (*ConfigLoader* method), 83
`get_amplitudes()` (*MultiConfig* method), 87
`get_angle()` (*CalAngleData* method), 128
`get_angle_amp()` (*CovTenDecayNew* method), 63
`get_angle_amp()` (*DecayChain* method), 55
`get_angle_amp()` (*DecayGroup* method), 55
`get_angle_amp()` (*HelicityDecay* method), 57
`get_angle_g_ls()` (*HelicityDecay* method), 57
`get_angle_helicity_amp()` (*HelicityDecay* method), 57
`get_angle_ls_amp()` (*HelicityDecay* method), 57
`get_angle_vars()` (*PlotParams* method), 85
`get_args_value()` (*MultiConfig* method), 87
`get_barrier_factor()` (*HelicityDecay* method), 57
`get_barrier_factor()` (*ParticleBWRLS* method), 78
`get_barrier_factor()` (*ParticleMultiBWR* method), 79
`get_barrier_factor2()` (*HelicityDecay* method), 57
`get_barrier_factor2()` (*ParticleDecayLS* method), 79
`get_barrier_factor_mass()` (*HelicityDecay* method), 57
`get_base_map()` (*DecayChain* method), 55
`get_base_map()` (*DecayGroup* method), 56
`get_beta()` (*KmatrixSingleChannelParticle* method), 43
`get_beta()` (*KmatrixSplitLSParticle* method), 43
`get_beta()` (*ParticleKmatrix* method), 52
`get_bin_index()` (*InterpolationParticle* method), 76
`get_bin_weight()` (*Hist1D* method), 146
`get_bool_mask()` (*AdaptiveBound* method), 117
`get_bound_patch()` (*AdaptiveBound* method), 117
`get_bounds()` (*AdaptiveBound* method), 117
`get_bounds_data()` (*AdaptiveBound* method), 117
`get_bprime_coeff()` (in module *tf_pwa.breit_wigner*), 127
`get_cached_int()` (*ModelCachedInt* method), 116
`get_cached_shape_idx()` (*CachedShapeAmplitude-Model* method), 44
`get_cg_coef()` (in module *tf_pwa.cg*), 132
`get_cg_matrix()` (*Decay* method), 149
`get_cg_matrix()` (*HelicityDecay* method), 57
`get_chain()` (*ConfigLoader* method), 83
`get_chain_data()` (in module *tf_pwa.cal_angle*), 131
`get_chain_from_particle()` (*DecayGroup* method), 150
`get_chain_name()` (in module *tf_pwa.experimental.factor_system*), 96
`get_chain_property()` (*ConfigLoader* method), 83
`get_chain_property()` (in module *tf_pwa.config_loader.plot*), 89
`get_chain_property_v1()` (in module *tf_pwa.config_loader.plot*), 89
`get_chain_property_v2()` (in module *tf_pwa.config_loader.plot*), 89
`get_chains_map()` (*DecayGroup* method), 150
`get_coeff()` (*ParticleFlate2* method), 65
`get_coeff()` (*ParticleFlateGen* method), 66
`get_config()` (in module *tf_pwa.config*), 132
`get_constrain_grad()` (*ConstrainModel* method), 108
`get_constrain_grad()` (*GaussianConstr* method), 110
`get_constrain_hessian()` (*ConstrainModel* method), 108
`get_constrain_hessian()` (*GaussianConstr* method), 110
`get_constrain_term()` (*ConstrainModel* method), 108
`get_constrain_term()` (*GaussianConstr* method), 110
`get_count()` (*Hist1D* method), 146
`get_cp_amp_total()` (*DecayChain* method), 55
`get_d2ydx2()` (*Bound* method), 157
`get_D_matrix_for_angle()` (in module *tf_pwa.dfun*), 139
`get_D_matrix_lambda()` (in module *tf_pwa.dfun*), 139
`get_dalitz()` (*ConfigLoader* method), 83
`get_dalitz()` (in module *tf_pwa.config_loader.plot*), 89
`get_dalitz_boundary()` (*ConfigLoader* method), 83
`get_dalitz_boundary()` (in module *tf_pwa.config_loader.plot*), 89
`get_dat_order()` (*ConfigLoader* method), 83
`get_dat_order()` (*SimpleData* method), 85
`get_data()` (*ConfigLoader* method), 83

`get_data()` (*MultiData method*), 85
`get_data()` (*MultiNpzData method*), 95
`get_data()` (*NpzData method*), 95
`get_data()` (*RootData method*), 86
`get_data()` (*SimpleData method*), 85
`get_data_file()` (*ConfigLoader method*), 83
`get_data_file()` (*SimpleData method*), 85
`get_data_index()` (*ConfigLoader method*), 83
`get_data_index()` (*PlotParams method*), 85
`get_data_index()` (*SimpleData method*), 85
`get_data_rec()` (*ConfigLoader method*), 83
`get_decay()` (*CalAngleData method*), 128
`get_decay()` (*ConfigLoader method*), 83
`get_decay()` (*DecayConfig method*), 87
`get_decay()` (*in module tf_pwa.amp.core*), 60
`get_decay()` (*in module tf_pwa.dec_parser*), 137
`get_decay_chain()` (*DecayGroup method*), 151
`get_decay_chain()` (*in module tf_pwa.amp.core*), 60
`get_decay_layout()` (*in module tf_pwa.vis*), 165
`get_decay_model()` (*in module tf_pwa.amp.core*), 60
`get_decay_struct()` (*DecayConfig method*), 87
`get_density_matrix()` (*DecayGroup method*), 56
`get_dot_source()` (*DotGenerator method*), 165
`get_dydx()` (*Bound method*), 158
`get_e()` (*LorentzVector method*), 120
`get_error()` (*ParamsTrans method*), 147
`get_error_matrix()` (*ParamsTrans method*), 147
`get_euler_angle()` (*SU2M method*), 120
`get_factor()` (*DecayChain method*), 55
`get_factor()` (*DecayGroup method*), 56
`get_factor()` (*HelicityDecay method*), 58
`get_factor()` (*HelicityDecayNP method*), 46
`get_factor()` (*Particle method*), 60
`get_factor_angle_amp()` (*DecayChain method*), 55
`get_factor_angle_amp()` (*DecayGroup method*), 56
`get_factor_angle_amp()` (*HelicityDecay method*), 58
`get_factor_angle_helicity_amp()` (*HelicityDecay method*), 58
`get_factor_H()` (*HelicityDecay method*), 58
`get_factor_H()` (*HelicityDecayNP method*), 46
`get_factor_m_dep()` (*HelicityDecay method*), 58
`get_factor_variable()` (*AmpBase method*), 54
`get_factor_variable()` (*DecayChain method*), 55
`get_factor_variable()` (*DecayGroup method*), 56
`get_factor_variable()` (*HelicityDecay method*), 58
`get_fcn()` (*ConfigLoader method*), 83
`get_fcn()` (*MultiConfig method*), 87
`get_fcns()` (*MultiConfig method*), 88
`get_finals_amp()` (*CovTenDecayChain method*), 63
`get_frac()` (*FitFractions method*), 144
`get_frac_diag_sum()` (*FitFractions method*), 144
`get_frac_grad()` (*FitFractions method*), 144
`get_func()` (*Bound method*), 158
`get_g_ls()` (*HelicityDecay method*), 58
`get_g_ls()` (*HelicityDecayCPV method*), 45
`get_gen()` (*ChainGenerator method*), 152
`get_gi()` (*KmatrixSingleChannelParticle method*), 43
`get_gi()` (*KmatrixSplitLSParticle method*), 43
`get_gi_frac()` (*KmatrixSplitLSParticle method*), 43
`get_gpu_free_memory()` (*in module tf_pwa.gpu_info*), 145
`get_gpu_info()` (*in module tf_pwa.gpu_info*), 145
`get_gpu_total_memory()` (*in module tf_pwa.gpu_info*), 145
`get_gpu_used_memory()` (*in module tf_pwa.gpu_info*), 145
`get_grad()` (*CombineFCN method*), 107
`get_grad()` (*FCN method*), 109
`get_grad()` (*ParamsTrans method*), 147
`get_grad_hessp()` (*CombineFCN method*), 107
`get_grad_hessp()` (*FCN method*), 109
`get_H()` (*HelicityDecayNP method*), 46
`get_H_barrier_factor()` (*HelicityDecayNPbf method*), 46
`get_H_zero_mask()` (*HelicityDecayNP method*), 46
`get_helicity_amp()` (*HelicityDecay method*), 58
`get_helicity_amp()` (*HelicityDecayNP method*), 46
`get_helicity_amp()` (*HelicityDecayNPbf method*), 46
`get_helicity_amp()` (*HelicityDecayP method*), 46
`get_histogram()` (*Frame method*), 90
`get_histogram()` (*PlotData method*), 90
`get_histogram()` (*PlotDataGroup method*), 90
`get_id()` (*BaseDecay method*), 147
`get_id()` (*DecayChain method*), 149
`get_id_swap_transpose()` (*DecayGroup method*), 56
`get_id_variable()` (*in module tf_pwa.experimental.factor_system*), 96
`get_key_content()` (*in module tf_pwa.cal_angle*), 131
`get_keys()` (*in module tf_pwa.cal_angle*), 131
`get_l_list()` (*Decay method*), 149
`get_label()` (*Plotter method*), 91
`get_layout()` (*in module tf_pwa.vis*), 165
`get_ls_amp()` (*HelicityDecay method*), 58
`get_ls_amp()` (*HelicityDecayCPV method*), 45
`get_ls_amp()` (*HelicityDecayNP method*), 46
`get_ls_amp()` (*HelicityDecayNPbf method*), 46
`get_ls_amp()` (*KmatrixSimple method*), 77
`get_ls_amp()` (*KmatrixSplitLSParticle method*), 43
`get_ls_amp()` (*ParticleBWRLS method*), 78
`get_ls_amp()` (*ParticleBWRLS2 method*), 79
`get_ls_amp()` (*ParticleDecay method*), 50
`get_ls_amp()` (*ParticleDecayLSKmatrix method*), 43
`get_ls_amp()` (*ParticleLS method*), 79
`get_ls_amp()` (*ParticleMultiBWR method*), 79
`get_ls_amp_frac()` (*ParticleBWRLS method*), 78
`get_ls_amp_org()` (*HelicityDecay method*), 58
`get_ls_list()` (*Decay method*), 149
`get_ls_list()` (*HelicityDecay method*), 58

`get_m_dep()` (*DecayChain* method), 55
`get_m_dep()` (*DecayGroup* method), 56
`get_m_dep()` (*HelicityDecay* method), 58
`get_m_dep_list()` (*CovTenDecayChain* method), 63
`get_mass()` (*CalAngleData* method), 128
`get_mass()` (*Particle* method), 60
`get_mass_range()` (*HelicityAngle* method), 94
`get_mass_range()` (*PhaseSpaceGenerator* method), 153
`get_mass_vars()` (*PlotParams* method), 85
`get_matrix_interp1d3()` (in module *tf_pwa.amp.interpolation*), 76
`get_matrix_interp1d3_v2()` (in module *tf_pwa.amp.interpolation*), 76
`get_max()` (*Seq* method), 142
`get_metric()` (*LorentzVector* method), 120
`get_mi()` (*KmatrixSingleChannelParticle* method), 43
`get_mi()` (*KmatrixSplitLSParticle* method), 43
`get_min_l()` (*Decay* method), 149
`get_momentum()` (*CalAngleData* method), 128
`get_n_data()` (*MultiData* method), 85
`get_n_data()` (*SimpleData* method), 85
`get_name()` (in module *tf_pwa.amp.core*), 60
`get_ndf()` (*ConfigLoader* method), 83
`get_nll()` (*CombineFCN* method), 107
`get_nll()` (*FCN* method), 109
`get_nll_grad()` (*CombineFCN* method), 107
`get_nll_grad()` (*FCN* method), 109
`get_nll_grad()` (*MixLogLikelihoodFCN* method), 110
`get_nll_grad_hessian()` (*CombineFCN* method), 108
`get_nll_grad_hessian()` (*FCN* method), 109
`get_node_layout()` (in module *tf_pwa.vis*), 165
`get_num_var()` (*Particle* method), 60
`get_num_var()` (*ParticleBW* method), 47
`get_num_var()` (*ParticleBWRLS* method), 78
`get_num_var()` (*ParticleFlateGen* method), 68
`get_num_var()` (*ParticleFlatte* method), 68
`get_p()` (in module *tf_pwa.phasespace*), 153
`get_p4()` (*RootData* method), 86
`get_params()` (*AbsPDF* method), 44
`get_params()` (*BaseModel* method), 106
`get_params()` (*CombineFCN* method), 108
`get_params()` (*ConfigLoader* method), 83
`get_params()` (*FCN* method), 110
`get_params()` (*Model* method), 111
`get_params()` (*MultiConfig* method), 88
`get_params()` (*PlotParams* method), 85
`get_params_error()` (*ConfigLoader* method), 83
`get_params_error()` (*MultiConfig* method), 88
`get_params_head()` (*AmpBase* method), 54
`get_params_head()` (*AmpDecay* method), 54
`get_params_head()` (*AmpDecayChain* method), 54
`get_params_head()` (*HelicityDecay* method), 58
`get_parity_term()` (in module *tf_pwa.amp.base*), 52
`get_particle()` (*DecayGroup* method), 151
`get_particle()` (in module *tf_pwa.amp.core*), 60
`get_particle_decay()` (*DecayChain* method), 150
`get_particle_function()` (*ConfigLoader* method), 83
`get_particle_function()` (in module *tf_pwa.config_loader.particle_function*), 88
`get_particle_model()` (in module *tf_pwa.amp.core*), 60
`get_particle_model_name()` (in module *tf_pwa.amp.core*), 60
`get_particle_p()` (*NpzData* method), 96
`get_particles()` (in module *tf_pwa.dec_parser*), 137
`get_phsp_factor()` (*HelicityAngle* method), 94
`get_phsp_factor()` (*HelicityAngle1* method), 95
`get_phsp_generator()` (*ConfigLoader* method), 83
`get_phsp_generator()` (in module *tf_pwa.config_loader.sample*), 93
`get_phsp_noeff()` (*ConfigLoader* method), 83
`get_phsp_noeff()` (*MultiData* method), 85
`get_phsp_noeff()` (*MultiNpzData* method), 95
`get_phsp_noeff()` (*SimpleData* method), 85
`get_phsp_p_generator()` (*ConfigLoader* method), 83
`get_phsp_p_generator()` (in module *tf_pwa.config_loader.sample*), 93
`get_phsp_plot()` (*ConfigLoader* method), 83
`get_phsp_plot()` (*SimpleData* method), 86
`get_plot_style()` (*Plotter* method), 91
`get_plotter()` (*ConfigLoader* method), 83
`get_plotter()` (in module *tf_pwa.config_loader.plotter*), 92
`get_point_values()` (*InterpLinearNpy* method), 73
`get_point_values()` (*InterpolationParticle* method), 76
`get_prod_chain()` (in module *tf_pwa.experimental.factor_system*), 96
`get_relative_momentum()` (*HelicityDecay* method), 58
`get_relative_momentum()` (in module *tf_pwa.cal_angle*), 131
`get_relative_momentum2()` (*HelicityDecay* method), 58
`get_relative_p()` (in module *tf_pwa.amp.core*), 62
`get_relative_p()` (in module *tf_pwa.amp.Kmatrix*), 43
`get_relative_p()` (in module *tf_pwa.amp.kmatrix_simple*), 77
`get_relative_p()` (in module *tf_pwa.formula*), 145
`get_relative_p2()` (in module *tf_pwa.amp.core*), 62
`get_relative_p2()` (in module *tf_pwa.formula*), 145
`get_res_map()` (*DecayGroup* method), 56
`get_res_style()` (*Plotter* method), 91
`get_resonances()` (*BaseParticle* method), 148

- `get_SDP_generator()` (*ConfigLoader* method), 83
`get_SDP_generator()` (in module *tf_pwa.config_loader.sample*), 93
`get_SDP_p_generator()` (*ConfigLoader* method), 83
`get_SDP_p_generator()` (in module *tf_pwa.config_loader.sample*), 93
`get_SDP_p_generator_legacy()` (in module *tf_pwa.config_loader.sample*), 93
`get_shape()` (in module *tf_pwa.model.model*), 114
`get_split_chain()` (in module *tf_pwa.experimental.factor_system*), 96
`get_style()` (*LineStyleSet* method), 88
`get_subdecay_mass()` (*Particle* method), 60
`get_swap_factor()` (*DecayGroup* method), 56
`get_swap_transpose()` (*DecayGroup* method), 56
`get_sympy_dom()` (*Particle* method), 60
`get_sympy_dom()` (*ParticleBWRCoupling* method), 49
`get_sympy_dom()` (*ParticleBWRLS* method), 78
`get_sympy_dom()` (*ParticleFlatteGen* method), 68
`get_sympy_dom()` (*ParticleFlatte* method), 68
`get_sympy_var()` (*Particle* method), 60
`get_sympy_var()` (*ParticleBW* method), 47
`get_sympy_var()` (*ParticleBWRLS* method), 78
`get_sympy_var()` (*ParticleFlatte* method), 69
`get_T()` (*LorentzVector* method), 119
`get_total_ls_list()` (*HelicityDecay* method), 58
`get_var()` (*AmpBase* method), 54
`get_variable_name()` (*AmpBase* method), 54
`get_weight()` (*CalAngleData* method), 128
`get_weight()` (*LazyCall* method), 135
`get_weight()` (*PhaseSpaceGenerator* method), 153
`get_weight()` (*RootData* method), 86
`get_weight_data()` (*Model* method), 111
`get_weight_data()` (*Model_new* method), 113
`get_weight_sign()` (*SimpleData* method), 86
`get_weight_smeasr()` (in module *tf_pwa.weight_smeasr*), 166
`get_width()` (*Particle* method), 60
`get_width()` (*ParticleFlatte* method), 69
`get_words()` (in module *tf_pwa.dec_parser*), 137
`get_X()` (*LorentzVector* method), 120
`get_X()` (*Vector3* method), 121
`get_x2y()` (*Bound* method), 158
`get_Y()` (*LorentzVector* method), 120
`get_Y()` (*Vector3* method), 121
`get_y2x()` (*Bound* method), 158
`get_Z()` (*LorentzVector* method), 120
`get_Z()` (*Vector3* method), 121
`get_zero_index()` (*HelicityDecayNP* method), 46
`GetA2BC_LS_list()` (in module *tf_pwa.particle*), 151
`Getp()` (in module *tf_pwa.cal_angle*), 128
`Getp2()` (in module *tf_pwa.cal_angle*), 129
`gls_combine()` (in module *tf_pwa.experimental.opt_int*), 97
`grad()` (*Cached_FG* method), 142
`grad()` (*CombineFCN* method), 108
`grad()` (*FCN* method), 110
`grad_hessp()` (*CombineFCN* method), 108
`grad_hessp()` (*FCN* method), 110
`grad_hessp_batch()` (*BaseModel* method), 106
`grad_hessp_batch()` (*Model* method), 111
`grad_hessp_batch()` (*ModelCachedAmp* method), 115
`GS()` (in module *tf_pwa.breit_wigner*), 126
- ## H
- `HeavyCall` (class in *tf_pwa.data*), 134
`HelicityAngle` (class in *tf_pwa.data_trans.helicity_angle*), 94
`HelicityAngle1` (class in *tf_pwa.data_trans.helicity_angle*), 94
`HelicityDecay` (class in *tf_pwa.amp.core*), 56
`HelicityDecayCPV` (class in *tf_pwa.amp.base*), 45
`HelicityDecayNP` (class in *tf_pwa.amp.base*), 45
`HelicityDecayNPbf` (class in *tf_pwa.amp.base*), 46
`HelicityDecayP` (class in *tf_pwa.amp.base*), 46
`hFun()` (in module *tf_pwa.breit_wigner*), 127
`Hist1D` (class in *tf_pwa.histogram*), 145
`hist_error()` (in module *tf_pwa.config_loader.plot*), 89
`hist_line()` (in module *tf_pwa.config_loader.plot*), 89
`hist_line_step()` (in module *tf_pwa.config_loader.plot*), 89
`histogram()` (*Hist1D* static method), 146
`HistParticle` (class in *tf_pwa.amp.interpolation*), 70
- ## I
- `identical_particles_swap()` (in module *tf_pwa.cal_angle*), 131
`identical_particles_swap_p()` (in module *tf_pwa.cal_angle*), 131
`index_generator()` (in module *tf_pwa.amp.core*), 62
`IndexMap` (class in *tf_pwa.amp.cov_ten*), 64
`infer_momentum()` (in module *tf_pwa.cal_angle*), 131
`init_name_list()` (*Variable* method), 159
`init_params()` (*AngSam3Decay* method), 54
`init_params()` (*BaseAmplitudeModel* method), 44
`init_params()` (*CovTenDecayChain* method), 63
`init_params()` (*CovTenDecaySimple* method), 64
`init_params()` (*DecayChain* method), 55
`init_params()` (*DecayGroup* method), 56
`init_params()` (*HelicityDecay* method), 58
`init_params()` (*HelicityDecayCPV* method), 45
`init_params()` (*HelicityDecayNP* method), 46
`init_params()` (*HelicityDecayNPbf* method), 46
`init_params()` (*HelicityDecayP* method), 46
`init_params()` (*Interp1DSpline* method), 70
`init_params()` (*Interp1DSplineIdx* method), 71
`init_params()` (*InterpLinearNpy* method), 73
`init_params()` (*InterpolationParticle* method), 76

- init_params() (*InterpSPPCHIP method*), 73
 init_params() (*KmatrixSimple method*), 77
 init_params() (*KmatrixSingleChannelParticle method*), 43
 init_params() (*KmatrixSplitLSParticle method*), 43
 init_params() (*Particle method*), 60
 init_params() (*ParticleBWRLS method*), 78
 init_params() (*ParticleDecayLS method*), 79
 init_params() (*ParticleDecayLSKmatrix method*), 43
 init_params() (*ParticleExp method*), 50
 init_params() (*ParticleExpCom method*), 50
 init_params() (*ParticleFlatte method*), 69
 init_params() (*ParticleKmatrix method*), 52
 init_params() (*ParticleLass method*), 52
 init_params() (*ParticleMultiBWR method*), 79
 init_params() (*ParticleOne method*), 52
 init_res_table() (*FitFractions method*), 144
 insert_extra_axis() (*EinSum method*), 64
 integral() (*BWGenerator method*), 97
 integral() (*FitFractions method*), 144
 integral() (*LinearInterp method*), 99
 Interp (*class in tf_pwa.amp.interpolation*), 70
 interp() (*Interp method*), 70
 interp() (*Interp1D3 method*), 70
 interp() (*Interp1DLang method*), 70
 interp() (*Interp1DSpline method*), 70
 interp() (*Interp1DSplineIdx method*), 71
 interp() (*InterpHist method*), 71
 interp() (*InterpHistIdx method*), 72
 interp() (*InterpL3 method*), 72
 interp() (*InterpLinearNpy method*), 73
 interp() (*InterpolationParticle method*), 76
 interp() (*InterpSPPCHIP method*), 73
 Interp1D3 (*class in tf_pwa.amp.interpolation*), 70
 interp1d3() (*in module tf_pwa.amp.interpolation*), 76
 Interp1DLang (*class in tf_pwa.amp.interpolation*), 70
 Interp1DSpline (*class in tf_pwa.amp.interpolation*), 70
 Interp1DSplineIdx (*class in tf_pwa.amp.interpolation*), 70
 Interp2D (*class in tf_pwa.generator.plane_2d*), 99
 interp_f() (*InterpNDHist method*), 98
 interp_hist() (*in module tf_pwa.histogram*), 146
 interp_sample() (*in module tf_pwa.generator.linear_interpolation*), 99
 interp_sample_f() (*in module tf_pwa.generator.linear_interpolation*), 99
 interp_sample_once() (*in module tf_pwa.generator.linear_interpolation*), 99
 InterpHist (*class in tf_pwa.amp.interpolation*), 71
 InterpHistIdx (*class in tf_pwa.amp.interpolation*), 71
 InterpL3 (*class in tf_pwa.amp.interpolation*), 72
 InterpLinearNpy (*class in tf_pwa.amp.interpolation*), 72
 InterpLinearTxt (*class in tf_pwa.amp.interpolation*), 73
 InterpND (*class in tf_pwa.generator.interp_nd*), 98
 InterpNDHist (*class in tf_pwa.generator.interp_nd*), 98
 InterpolationParticle (*class in tf_pwa.amp.interpolation*), 76
 InterpSPPCHIP (*class in tf_pwa.amp.interpolation*), 73
 intgral_step() (*InterpND method*), 98
 intgral_step() (*InterpNDHist method*), 98
 inv() (*SU2M method*), 120
 inverse() (*BaseTransform method*), 155
 inverse() (*LinearTrans method*), 155
 is_complex() (*in module tf_pwa.utils*), 156
 is_fixed() (*Variable method*), 159
 is_fixed_shape() (*Particle method*), 60
 is_fixed_shape() (*ParticleLS method*), 79
- ## J
- json_print() (*in module tf_pwa.app.fit*), 81
- ## K
- kine_max() (*in module tf_pwa.angle*), 121
 kine_min() (*in module tf_pwa.angle*), 121
 kine_min_max() (*in module tf_pwa.angle*), 121
 KMatrix_single() (*in module tf_pwa.amp.Kmatrix*), 41
 KmatrixSimple (*class in tf_pwa.amp.kmatrix_simple*), 77
 KmatrixSingleChannelParticle (*class in tf_pwa.amp.Kmatrix*), 41
 KmatrixSplitLSParticle (*class in tf_pwa.amp.Kmatrix*), 43
- ## L
- LargeNumberError, 141
 LazyCall (*class in tf_pwa.data*), 134
 LazyFile (*class in tf_pwa.data*), 135
 likelihood_profile() (*ConfigLoader method*), 84
 likelihood_profile() (*in module tf_pwa.applications*), 124
 line_search_nonmonote() (*in module tf_pwa.fit_improve*), 142
 line_search_wolfe2() (*in module tf_pwa.fit_improve*), 142
 LinearInterp (*class in tf_pwa.generator.linear_interpolation*), 99
 LinearInterpImportance (*class in tf_pwa.generator.linear_interpolation*), 99
 LinearTrans (*class in tf_pwa.transform*), 155
 LineSearchWarning, 142
 LineStyleSet (*class in tf_pwa.config_loader.plot*), 88
 list_helicity_inner() (*AmpDecay method*), 54
 list_to_tuple() (*in module tf_pwa.amp.preprocess*), 78
 load_cached_data() (*ConfigLoader method*), 84

- load_cached_data() (*SimpleData* method), 86
 load_config() (*ConfigLoader* static method), 84
 load_config() (*DecayConfig* static method), 87
 load_config() (in module *tf_pwa.config_loader.base_config*), 81
 load_config_file() (in module *tf_pwa.utils*), 156
 load_dat_file() (in module *tf_pwa.data*), 136
 load_data() (in module *tf_pwa.data*), 137
 load_data() (*NpzData* method), 96
 load_data() (*SimpleData* method), 86
 load_data_mode() (in module *tf_pwa.config_loader.data*), 86
 load_dec() (in module *tf_pwa.dec_parser*), 138
 load_dec_file() (in module *tf_pwa.dec_parser*), 138
 load_decfile_particle() (in module *tf_pwa.amp.core*), 62
 load_extra_var() (*SimpleData* method), 86
 load_p4() (*SimpleData* method), 86
 load_root_data() (in module *tf_pwa.root_io*), 154
 load_Ttree() (in module *tf_pwa.root_io*), 154
 load_var() (*RootData* method), 86
 load_weight_file() (*SimpleData* method), 86
 log() (*NumberError* method), 140
 loop_split_bound() (*AdaptiveBound* static method), 117
 lorentz_neg() (in module *tf_pwa.data_trans.helicity_angle*), 95
 LorentzVector (class in *tf_pwa.angle*), 119
 ls_selector_qr() (in module *tf_pwa.amp.core*), 62
 ls_selector_weight() (in module *tf_pwa.cov_ten_ir*), 133
- ## M
- M() (*LorentzVector* method), 119
 M2() (*LorentzVector* method), 119
 mask_factor_vars() (*HelicityDecay* method), 58
 mask_params() (*AbsPDF* method), 44
 mask_params() (*ConfigLoader* method), 84
 mask_params() (*ParamsTrans* method), 147
 mask_params() (*VarsManager* method), 162
 mass() (*ParticleMultiBWR* method), 79
 mass2() (in module *tf_pwa.amp.cov_ten*), 64
 mass_hist() (*CalAngleData* method), 128
 mass_importances() (*PhaseSpaceGenerator* method), 153
 mass_linspace() (*HelicityAngle* method), 94
 mass_linspace() (*ParticleFunction* method), 88
 mass_range() (*ParticleFunction* method), 88
 MassiveTransAngle() (in module *tf_pwa.cov_ten_ir*), 133
 MasslessTransAngle() (in module *tf_pwa.cov_ten_ir*), 133
 merge() (*LazyCall* method), 135
 merge_hist() (in module *tf_pwa.config_loader.plotter*), 92
 minimize() (in module *tf_pwa.fit_improve*), 143
 minimize() (*VarsManager* method), 162
 minimize_error() (*VarsManager* method), 162
 mix_data_bakcground() (*Model* method), 111
 MixLogLikelihoodFCN (class in *tf_pwa.model.model*), 110
 Model (class in *tf_pwa.model.model*), 110
 Model_cfit (class in *tf_pwa.model.cfit*), 103
 Model_cfit_cached (class in *tf_pwa.model.cfit*), 103
 model_name (*AngSam3Decay* attribute), 54
 model_name (*CovTenDecayChain* attribute), 63
 model_name (*CovTenDecaySimple* attribute), 64
 model_name (*DecayChain* attribute), 55
 model_name (*HelicityDecay* attribute), 58
 model_name (*HelicityDecayCPV* attribute), 45
 model_name (*HelicityDecayNP* attribute), 46
 model_name (*HelicityDecayNPbf* attribute), 46
 model_name (*HelicityDecayP* attribute), 47
 model_name (*Interp* attribute), 70
 model_name (*Interp1D3* attribute), 70
 model_name (*Interp1DLang* attribute), 70
 model_name (*Interp1DSpline* attribute), 70
 model_name (*Interp1DSplineIdx* attribute), 71
 model_name (*InterpHist* attribute), 71
 model_name (*InterpHistIdx* attribute), 72
 model_name (*InterpL3* attribute), 72
 model_name (*InterpLinearNpy* attribute), 73
 model_name (*InterpLinearTxt* attribute), 73
 model_name (*InterpSPPCHIP* attribute), 76
 model_name (*KmatrixSimple* attribute), 77
 model_name (*KmatrixSingleChannelParticle* attribute), 43
 model_name (*KmatrixSplitLSParticle* attribute), 43
 model_name (*Particle* attribute), 60
 model_name (*ParticleBW* attribute), 47
 model_name (*ParticleBWR2* attribute), 48
 model_name (*ParticleBWR_normal* attribute), 49
 model_name (*ParticleBWRBelowThreshold* attribute), 48
 model_name (*ParticleBWRCoupling* attribute), 49
 model_name (*ParticleBWRLS* attribute), 78
 model_name (*ParticleBWRLS2* attribute), 79
 model_name (*ParticleDecay* attribute), 50
 model_name (*ParticleDecayLS* attribute), 79
 model_name (*ParticleDecayLSKmatrix* attribute), 43
 model_name (*ParticleExp* attribute), 50
 model_name (*ParticleExpCom* attribute), 50
 model_name (*ParticleFlate2* attribute), 65
 model_name (*ParticleFlateGen* attribute), 68
 model_name (*ParticleFlatte* attribute), 69
 model_name (*ParticleFlatteC* attribute), 69
 model_name (*ParticleGS* attribute), 52
 model_name (*ParticleKmatrix* attribute), 52
 model_name (*ParticleLass* attribute), 52

model_name (*ParticleMultiBW attribute*), 79
 model_name (*ParticleMultiBWR attribute*), 79
 model_name (*ParticleOne attribute*), 52
 model_name (*ParticleX attribute*), 60
 Model_new (*class in tf_pwa.model.model*), 112
 ModelCachedAmp (*class in tf_pwa.model.opt_int*), 115
 ModelCachedInt (*class in tf_pwa.model.opt_int*), 116
 ModelCfitExtended (*class in tf_pwa.model.cfit*), 102
 module
 tf_pwa, 41
 tf_pwa.adaptive_bins, 117
 tf_pwa.amp, 41
 tf_pwa.amp.amp, 44
 tf_pwa.amp.base, 45
 tf_pwa.amp.core, 54
 tf_pwa.amp.cov_ten, 63
 tf_pwa.amp.flatte, 65
 tf_pwa.amp.interpolation, 70
 tf_pwa.amp.Kmatrix, 41
 tf_pwa.amp.kmatrix_simple, 77
 tf_pwa.amp.preprocess, 77
 tf_pwa.amp.split_ls, 78
 tf_pwa.angle, 118
 tf_pwa.app, 81
 tf_pwa.app.fit, 81
 tf_pwa.applications, 121
 tf_pwa.breit_wigner, 125
 tf_pwa.cal_angle, 127
 tf_pwa.cg, 132
 tf_pwa.config, 132
 tf_pwa.config_loader, 81
 tf_pwa.config_loader.base_config, 81
 tf_pwa.config_loader.config_loader, 81
 tf_pwa.config_loader.data, 85
 tf_pwa.config_loader.data_root_lhcb, 86
 tf_pwa.config_loader.decay_config, 87
 tf_pwa.config_loader.extra, 87
 tf_pwa.config_loader.multi_config, 87
 tf_pwa.config_loader.particle_function, 88
 tf_pwa.config_loader.plot, 88
 tf_pwa.config_loader.plotter, 90
 tf_pwa.config_loader.sample, 93
 tf_pwa.cov_ten_ir, 133
 tf_pwa.data, 133
 tf_pwa.data_trans, 94
 tf_pwa.data_trans.dalitz, 94
 tf_pwa.data_trans.helicity_angle, 94
 tf_pwa.dec_parser, 137
 tf_pwa.dfun, 138
 tf_pwa.einsum, 140
 tf_pwa.err_num, 140
 tf_pwa.experimental, 95
 tf_pwa.experimental.build_amp, 95
 tf_pwa.experimental.extra_amp, 95
 tf_pwa.experimental.extra_data, 95
 tf_pwa.experimental.extra_function, 96
 tf_pwa.experimental.factor_system, 96
 tf_pwa.experimental.opt_int, 97
 tf_pwa.experimental.wrap_function, 97
 tf_pwa.experimental, 141
 tf_pwa.fit, 141
 tf_pwa.fit_improve, 142
 tf_pwa.fitfractions, 144
 tf_pwa.formula, 145
 tf_pwa.function, 145
 tf_pwa.generator, 97
 tf_pwa.generator.breit_wigner, 97
 tf_pwa.generator.generator, 98
 tf_pwa.generator.interp_nd, 98
 tf_pwa.generator.linear_interpolation, 99
 tf_pwa.generator.plane_2d, 99
 tf_pwa.generator.square_dalitz_plot, 100
 tf_pwa.gpu_info, 145
 tf_pwa.histogram, 145
 tf_pwa.main, 147
 tf_pwa.model, 102
 tf_pwa.model.cfit, 102
 tf_pwa.model.custom, 104
 tf_pwa.model.model, 106
 tf_pwa.model.opt_int, 115
 tf_pwa.params_trans, 147
 tf_pwa.particle, 147
 tf_pwa.phasespace, 152
 tf_pwa.root_io, 154
 tf_pwa.significance, 154
 tf_pwa.tensorflow_wrapper, 154
 tf_pwa.transform, 155
 tf_pwa.utils, 155
 tf_pwa.variable, 157
 tf_pwa.version, 165
 tf_pwa.vis, 165
 tf_pwa.weight_smear, 166
 Module (*class in tf_pwa.tensorflow_wrapper*), 154
 multi_sampling() (*in tf_pwa.generator.generator*), 98
 multi_split_bound() (*AdaptiveBound static method*), 117
 MultiConfig (*class in tf_pwa.config_loader.multi_config*), 87
 MultiData (*class in tf_pwa.config_loader.data*), 85
 MultiNpzData (*class in tf_pwa.experimental.extra_data*), 95

N
 n_helicity_inner() (*AmpDecay method*), 54
 n_points() (*HistParticle method*), 70
 n_points() (*InterpolationParticle method*), 76

name (*BaseDecay* property), 147
 name (*BaseParticle* property), 148
 ndf() (*Hist1D* method), 146
 neg() (*LorentzVector* method), 120
 nll() (*BaseCustomModel* method), 104
 nll() (*BaseModel* method), 106
 nll() (*ConstrainModel* method), 108
 nll() (*Model* method), 111
 nll() (*Model_cfit* method), 103
 nll() (*Model_new* method), 113
 nll() (*ModelCfitExtended* method), 102
 nll_funciton() (in module *tf_pwa.function*), 145
 nll_grad() (*BaseModel* method), 106
 nll_grad() (*CombineFCN* method), 108
 nll_grad() (*FCN* method), 110
 nll_grad() (in module *tf_pwa.fitfractions*), 144
 nll_grad() (*Model* method), 112
 nll_grad_batch() (*BaseCustomModel* method), 104
 nll_grad_batch() (*BaseModel* method), 106
 nll_grad_batch() (*Model* method), 112
 nll_grad_batch() (*Model_cfit* method), 103
 nll_grad_batch() (*Model_cfit_cached* method), 104
 nll_grad_batch() (*Model_new* method), 113
 nll_grad_batch() (*ModelCachedAmp* method), 115
 nll_grad_batch() (*ModelCachedInt* method), 116
 nll_grad_batch() (*ModelCfitExtended* method), 102
 nll_grad_hessian() (*BaseCustomModel* method), 105
 nll_grad_hessian() (*BaseModel* method), 106
 nll_grad_hessian() (*CombineFCN* method), 108
 nll_grad_hessian() (*FCN* method), 110
 nll_grad_hessian() (*Model* method), 112
 nll_grad_hessian() (*Model_cfit* method), 103
 nll_grad_hessian() (*Model_new* method), 113
 nll_grad_hessian() (*ModelCachedInt* method), 116
 nll_grad_hessian() (*ModelCfitExtended* method), 102
 nll_gradient() (*ConstrainModel* method), 108
 norm() (*Vector3* method), 121
 norm2() (*Vector3* method), 121
 normal() (in module *tf_pwa.data_trans.helicity_angle*), 95
 normal_factor() (in module *tf_pwa.cov_ten_ir*), 133
 normal_quantile() (in module *tf_pwa.significance*), 154
 NpzData (class in *tf_pwa.experimental.extra_data*), 95
 num_hess_inv_3point() (in module *tf_pwa.applications*), 125
 NumberError (class in *tf_pwa.err_num*), 140
 numpy_cross() (in module *tf_pwa.tensorflow_wrapper*), 154
 NumSL() (in module *tf_pwa.cov_ten_ir*), 133
 NumSL0() (in module *tf_pwa.cov_ten_ir*), 133
 NumSL1() (in module *tf_pwa.cov_ten_ir*), 133
 NumSL2() (in module *tf_pwa.cov_ten_ir*), 133

NumSL3() (in module *tf_pwa.cov_ten_ir*), 133

O

old_style() (*Plotter* method), 91
 omega() (*LorentzVector* method), 120
 one() (in module *tf_pwa.breit_wigner*), 127
 opt_lambdaify() (in module *tf_pwa.amp.Kmatrix*), 43
 ordered_indices() (in module *tf_pwa.einsum*), 140

P

P4DirectlyAmplitudeModel (class in *tf_pwa.amp.amp*), 45
 params_trans() (*ConfigLoader* method), 84
 params_trans() (*MultiConfig* method), 88
 ParamsTrans (class in *tf_pwa.params_trans*), 147
 parity_trans() (in module *tf_pwa.cal_angle*), 131
 partial_amp() (in module *tf_pwa.experimental.factor_system*), 96
 partial_weight() (*AmplitudeModel* method), 44
 partial_weight() (*BaseAmplitudeModel* method), 44
 partial_weight() (*DecayGroup* method), 56
 partial_weight_interference() (*BaseAmplitudeModel* method), 44
 partial_weight_interference() (*DecayGroup* method), 56
 Particle (class in *tf_pwa.amp.core*), 58
 particle_item() (*DecayConfig* static method), 87
 particle_item_list() (*DecayConfig* static method), 87
 ParticleBW (class in *tf_pwa.amp.base*), 47
 ParticleBWR2 (class in *tf_pwa.amp.base*), 47
 ParticleBWR_normal (class in *tf_pwa.amp.base*), 49
 ParticleBWRBelowThreshold (class in *tf_pwa.amp.base*), 48
 ParticleBWRCoupling (class in *tf_pwa.amp.base*), 48
 ParticleBWRLS (class in *tf_pwa.amp.split_ls*), 78
 ParticleBWRLS2 (class in *tf_pwa.amp.split_ls*), 78
 ParticleDecay (class in *tf_pwa.amp.base*), 49
 ParticleDecayLS (class in *tf_pwa.amp.split_ls*), 79
 ParticleDecayLSKmatrix (class in *tf_pwa.amp.Kmatrix*), 43
 ParticleExp (class in *tf_pwa.amp.base*), 50
 ParticleExpCom (class in *tf_pwa.amp.base*), 50
 ParticleFlate2 (class in *tf_pwa.amp.flatte*), 65
 ParticleFlateGen (class in *tf_pwa.amp.flatte*), 65
 ParticleFlatte (class in *tf_pwa.amp.flatte*), 68
 ParticleFlatteC (class in *tf_pwa.amp.flatte*), 69
 ParticleFunction (class in *tf_pwa.config_loader.particle_function*), 88
 ParticleGS (class in *tf_pwa.amp.base*), 50
 ParticleKmatrix (class in *tf_pwa.amp.base*), 52
 ParticleLass (class in *tf_pwa.amp.base*), 52
 ParticleList (class in *tf_pwa.particle*), 151

- ParticleLS (class in *tf_pwa.amp.split_ls*), 79
 ParticleMultiBW (class in *tf_pwa.amp.split_ls*), 79
 ParticleMultiBWR (class in *tf_pwa.amp.split_ls*), 79
 ParticleOne (class in *tf_pwa.amp.base*), 52
 ParticleX (class in *tf_pwa.amp.core*), 60
 pdf() (*BaseAmplitudeModel* method), 44
 pdf() (*CachedAmpAmplitudeModel* method), 44
 pdf() (*CachedShapeAmplitudeModel* method), 45
 pdf() (*FactorAmplitudeModel* method), 45
 pdf() (*P4DirectlyAmplitudeModel* method), 45
 perfer_node() (in module *tf_pwa.config_loader.sample*), 93
 PhaseSpaceGenerator (class in *tf_pwa.phasespace*), 152
 phsp_factor() (*ParticleFunction* method), 88
 phsp_fractor() (*KmatrixSimple* method), 77
 phsp_fractor() (*ParticleFunction* method), 88
 plot_adaptive_2dpull() (*ConfigLoader* method), 84
 plot_adaptive_2dpull() (in module *tf_pwa.config_loader.plot*), 89
 plot_bound() (*AdaptiveBound* method), 118
 plot_decay_struct() (in module *tf_pwa.vis*), 165
 plot_frame() (*Plotter* method), 91
 plot_frame_with_pull() (*Plotter* method), 91
 plot_function_2dpull() (in module *tf_pwa.config_loader.plot*), 89
 plot_hist() (in module *tf_pwa.histogram*), 146
 plot_partial_wave() (*ConfigLoader* method), 84
 plot_partial_wave() (in module *tf_pwa.config_loader.plot*), 89
 plot_partial_wave() (*MultiConfig* method), 88
 plot_partial_wave_interf() (*ConfigLoader* method), 84
 plot_partial_wave_interf() (in module *tf_pwa.config_loader.plot*), 90
 plot_particle_model() (in module *tf_pwa.utils*), 156
 plot_pull() (in module *tf_pwa.applications*), 125
 plot_var() (*Plotter* method), 92
 PlotAllData (class in *tf_pwa.config_loader.plotter*), 90
 PlotData (class in *tf_pwa.config_loader.plotter*), 90
 PlotDataGroup (class in *tf_pwa.config_loader.plotter*), 90
 PlotParams (class in *tf_pwa.config_loader.config_loader*), 85
 Plotter (class in *tf_pwa.config_loader.plotter*), 90
 poisson_smear() (in module *tf_pwa.weight_smear*), 166
 pprint() (in module *tf_pwa.utils*), 156
 prepare_data_from_dat_file() (in module *tf_pwa.cal_angle*), 131
 prepare_data_from_dat_file4() (in module *tf_pwa.cal_angle*), 131
 prepare_data_from_decay() (in module *tf_pwa.cal_angle*), 131
 print_dic() (in module *tf_pwa.utils*), 156
 prob() (in module *tf_pwa.significance*), 154
 process_decay_card() (in module *tf_pwa.dec_parser*), 138
 process_scale() (*MultiData* method), 85
 process_scale() (*SimpleData* method), 86
 product_gls() (*DecayChain* method), 55
 PWFA() (in module *tf_pwa.cov_ten_ir*), 133
- ## R
- r_shareto() (*Variable* method), 159
 read() (*BaseTransform* method), 155
 read() (*VarsManager* method), 162
 ReadData (class in *tf_pwa.config_loader.plotter*), 92
 real_var() (*Variable* method), 159
 refresh_vars() (*VarsManager* method), 162
 regist_command() (in module *tf_pwa.dec_parser*), 138
 regist_config() (in module *tf_pwa.config*), 132
 regist_decay() (in module *tf_pwa.amp.core*), 62
 regist_function() (in module *tf_pwa.tensorflow_wrapper*), 154
 regist_lineshape() (in module *tf_pwa.breit_wigner*), 127
 regist_particle() (in module *tf_pwa.amp.core*), 62
 regist_subcommand() (in module *tf_pwa.main*), 147
 register_amp_model() (in module *tf_pwa.amp.amp*), 45
 register_data_mode() (in module *tf_pwa.config_loader.data*), 86
 register_decay() (in module *tf_pwa.amp.core*), 62
 register_decay_chain() (in module *tf_pwa.amp.core*), 62
 register_extra_constrains() (*ConfigLoader* method), 84
 register_function() (*ConfigLoader* class method), 84
 register_particle() (in module *tf_pwa.amp.core*), 62
 register_preprocessor() (in module *tf_pwa.amp.preprocess*), 78
 register_weight_smear() (in module *tf_pwa.weight_smear*), 166
 reinit_params() (*ConfigLoader* method), 84
 reinit_params() (*MultiConfig* method), 88
 remove_bound() (*VarsManager* method), 162
 remove_comment() (in module *tf_pwa.dec_parser*), 138
 remove_decay() (*BaseParticle* method), 148
 remove_size1() (in module *tf_pwa.einsum*), 140
 remove_var() (*VarsManager* method), 162
 rename() (*Variable* method), 160
 rename_data_dict() (in module *tf_pwa.amp.core*), 62
 rename_params() (*DecayConfig* method), 87
 rename_var() (*VarsManager* method), 162
 reorder_final_particle() (in module *tf_pwa.vis*), 165

- replace_ellipsis() (in module *tf_pwa.einsum*), 140
 replace_none_in_shape() (in module *tf_pwa.einsum*), 140
 required_params (SimpleCFitModel attribute), 105
 required_params (SimpleNllFracModel attribute), 105
 rest_vector() (LorentzVector method), 120
 reverse_bessel_polynomials() (in module *tf_pwa.breit_wigner*), 127
 reweight_init_value() (ConfigLoader static method), 85
 RootData (class in *tf_pwa.config_loader.data_root_lhcb*), 86
 Rotation_y() (SU2M static method), 120
 Rotation_z() (SU2M static method), 120
 rp2xy() (VarsManager method), 163
 rp2xy_all() (VarsManager method), 163
- ## S
- sameas() (Variable method), 160
 sample_test_function() (in module *tf_pwa.generator.linear_interpolation*), 99
 save() (LineStyleSet method), 88
 save() (StyleSet method), 92
 save_all_frame() (Plotter method), 92
 save_as() (FitResult method), 141
 save_cached_data() (ConfigLoader method), 85
 save_cached_data() (SimpleData method), 86
 save_data() (in module *tf_pwa.data*), 137
 save_dataz() (in module *tf_pwa.data*), 137
 save_dict_to_root() (in module *tf_pwa.root_io*), 154
 save_frac_csv() (in module *tf_pwa.utils*), 156
 save_params() (ConfigLoader method), 85
 save_params() (MultiConfig method), 88
 save_tensorflow_model() (ConfigLoader method), 85
 savetxt() (CalAngleData method), 128
 savetxt() (SimpleData method), 86
 scalar_search_wolfe2() (in module *tf_pwa.fit_improve*), 143
 scale_to() (Hist1D method), 146
 scale_to() (WeightedData method), 146
 SCombLS() (in module *tf_pwa.cov_ten_ir*), 133
 SDPGenerator (class in *tf_pwa.generator.square_dalitz_plot*), 100
 Seq (class in *tf_pwa.fit_improve*), 142
 set() (StyleSet method), 92
 set() (VarsManager method), 163
 set_all() (VarsManager method), 163
 set_axis() (Frame method), 90
 set_bound() (Variable method), 160
 set_bound() (VarsManager method), 163
 set_cached_file() (LazyCall method), 135
 set_config() (in module *tf_pwa.config*), 132
 set_decay() (PhaseSpaceGenerator method), 153
 set_error() (FitResult method), 141
 set_fix() (VarsManager method), 163
 set_fix_idx() (Variable method), 160
 set_gen() (GenTest method), 98
 set_gpu_mem_growth() (in module *tf_pwa.tensorflow_wrapper*), 154
 set_inputs() (EinSum method), 64
 set_lazy_call() (MultiData method), 85
 set_lazy_call() (SimpleData method), 86
 set_ls() (HelicityDecay method), 58
 set_min_max() (in module *tf_pwa.config_loader.decay_config*), 87
 set_name() (BaseParticle method), 148
 set_params() (AbsPDF method), 44
 set_params() (BaseModel method), 107
 set_params() (ConfigLoader method), 85
 set_params() (Model method), 112
 set_params() (MultiConfig method), 88
 set_phi() (Variable method), 160
 set_plot_item() (Plotter method), 92
 set_prefix_constrains() (in module *tf_pwa.config_loader.config_loader*), 85
 set_random_seed() (in module *tf_pwa.data*), 137
 set_rho() (Variable method), 160
 set_same() (VarsManager method), 163
 set_same_ratio() (Variable method), 160
 set_share_r() (VarsManager method), 163
 set_trans_var() (VarsManager method), 164
 set_used_chains() (BaseAmplitudeModel method), 44
 set_used_chains() (DecayGroup method), 56
 set_used_res() (BaseAmplitudeModel method), 44
 set_used_res() (DecayGroup method), 56
 set_value() (Variable method), 160
 sigle_decay() (in module *tf_pwa.dec_parser*), 138
 significance() (in module *tf_pwa.significance*), 154
 simple_cache_fun() (in module *tf_pwa.amp.core*), 62
 simple_cache_fun() (in module *tf_pwa.particle*), 151
 simple_deepcopy() (in module *tf_pwa.amp.core*), 63
 simple_resonance() (in module *tf_pwa.amp.core*), 63
 SimpleCFitModel (class in *tf_pwa.model.custom*), 105
 SimpleChi2Model (class in *tf_pwa.model.custom*), 105
 SimpleClipNllModel (class in *tf_pwa.model.custom*), 105
 SimpleData (class in *tf_pwa.config_loader.data*), 85
 SimpleNllFracModel (class in *tf_pwa.model.custom*), 105
 SimpleNllModel (class in *tf_pwa.model.custom*), 105
 SimpleResonances (class in *tf_pwa.amp.core*), 60
 single_sampling() (in module *tf_pwa.config_loader.sample*), 94
 single_sampling2() (in module *tf_pwa.generator.generator*), 98
 single_split_bound() (AdaptiveBound static method), 118

- small_d_matrix() (in module *tf_pwa.dfun*), 139
 small_d_weight() (in module *tf_pwa.dfun*), 140
 solve() (*BWGenerator* method), 97
 solve() (*LinearInterp* method), 99
 solve_2() (in module *tf_pwa.generator.plane_2d*), 100
 solve_3() (in module *tf_pwa.generator.plane_2d*), 100
 solve_left() (*TriangleGenerator* method), 100
 solve_pole() (*Particle* method), 60
 solve_right() (*TriangleGenerator* method), 100
 solve_s() (*TriangleGenerator* method), 100
 sorted_table() (*DecayChain* method), 150
 sorted_table_layers() (*DecayChain* method), 150
 sphericalHarmonic() (in module *tf_pwa.cov_ten_ir*), 133
 spline_matrix() (in module *tf_pwa.amp.interpolation*), 76
 spline_x_matrix() (in module *tf_pwa.amp.interpolation*), 76
 spline_xi_matrix() (in module *tf_pwa.amp.interpolation*), 76
 split_data() (*AdaptiveBound* method), 118
 split_full_data() (*AdaptiveBound* method), 118
 split_generator() (in module *tf_pwa.data*), 137
 split_gls() (in module *tf_pwa.experimental.opt_int*), 97
 split_len() (in module *tf_pwa.particle*), 152
 split_lines() (in module *tf_pwa.dec_parser*), 138
 split_particle_type() (in module *tf_pwa.particle*), 152
 split_particle_type_list() (in module *tf_pwa.particle*), 152
 sppchip() (in module *tf_pwa.amp.interpolation*), 76
 sppchip_coeffs() (in module *tf_pwa.amp.interpolation*), 76
 square_dalitz_cut() (in module *tf_pwa.generator.square_dalitz_plot*), 100
 square_dalitz_variables() (in module *tf_pwa.generator.square_dalitz_plot*), 100
 standard_complex() (*VarsManager* method), 164
 standard_topology() (*DecayChain* method), 150
 std_periodic_var() (in module *tf_pwa.utils*), 156
 std_polar() (in module *tf_pwa.utils*), 157
 std_polar() (*VarsManager* method), 164
 std_polar_all() (*VarsManager* method), 164
 strip_variable() (in module *tf_pwa.experimental.factor_system*), 96
 struct_momentum() (in module *tf_pwa.cal_angle*), 131
 StyleSet (class in *tf_pwa.config_loader.plotter*), 92
 SU2M (class in *tf_pwa.angle*), 120
 sum_amp() (*DecayGroup* method), 56
 sum_amp_polarization() (*DecayGroup* method), 56
 sum_grad_hessp() (in module *tf_pwa.model.model*), 114
 sum_grad_hessp_data2() (in module *tf_pwa.model.opt_int*), 116
 sum_gradient() (in module *tf_pwa.fitfractions*), 144
 sum_gradient() (in module *tf_pwa.model.model*), 114
 sum_gradient() (in module *tf_pwa.model.opt_int*), 116
 sum_gradient_data2() (in module *tf_pwa.model.opt_int*), 117
 sum_gradient_new() (in module *tf_pwa.model.model*), 114
 sum_hessian() (in module *tf_pwa.model.model*), 114
 sum_hessian_new() (in module *tf_pwa.model.model*), 114
 sum_log_integral_grad_batch() (*BaseModel* method), 107
 sum_log_integral_grad_batch() (*Model* method), 112
 sum_log_integral_grad_batch() (*Model-CachedAmp* method), 115
 sum_nll_grad_bacch() (*BaseModel* method), 107
 sum_nll_grad_bacch() (*Model* method), 112
 sum_nll_grad_bacch() (*ModelCachedAmp* method), 115
 sum_no_gradient() (in module *tf_pwa.fitfractions*), 144
 sum_resolution() (*BaseModel* method), 107
 sum_resolution() (*Model* method), 112
 sum_with_polarization() (*DecayGroup* method), 56
 SumVar (class in *tf_pwa.variable*), 158
 symbol_generate() (in module *tf_pwa.einsum*), 140
- ## T
- t_min_max() (*TriangleGenerator* method), 100
 temp_config() (in module *tf_pwa.config*), 132
 temp_params() (*AbsPDF* method), 44
 temp_params() (*VarsManager* method), 164
 temp_total_gls_one() (*BaseAmplitudeModel* method), 44
 temp_used_res() (*BaseAmplitudeModel* method), 44
 temp_used_res() (*DecayGroup* method), 56
 temp_var() (in module *tf_pwa.experimental.factor_system*), 96
 tensor_einsum_reduce_sum() (in module *tf_pwa.einsum*), 140
 tf_pwa module, 41
 tf_pwa.adaptive_bins module, 117
 tf_pwa.amp module, 41
 tf_pwa.amp.amp module, 44
 tf_pwa.amp.base module, 45
 tf_pwa.amp.core

- module, 54
- tf_pwa.amp.cov_ten
 - module, 63
- tf_pwa.amp.flatte
 - module, 65
- tf_pwa.amp.interpolation
 - module, 70
- tf_pwa.amp.Kmatrix
 - module, 41
- tf_pwa.amp.kmatrix_simple
 - module, 77
- tf_pwa.amp.preprocess
 - module, 77
- tf_pwa.amp.split_ls
 - module, 78
- tf_pwa.angle
 - module, 118
- tf_pwa.app
 - module, 81
- tf_pwa.app.fit
 - module, 81
- tf_pwa.applications
 - module, 121
- tf_pwa.breit_wigner
 - module, 125
- tf_pwa.cal_angle
 - module, 127
- tf_pwa.cg
 - module, 132
- tf_pwa.config
 - module, 132
- tf_pwa.config_loader
 - module, 81
- tf_pwa.config_loader.base_config
 - module, 81
- tf_pwa.config_loader.config_loader
 - module, 81
- tf_pwa.config_loader.data
 - module, 85
- tf_pwa.config_loader.data_root_lhcb
 - module, 86
- tf_pwa.config_loader.decay_config
 - module, 87
- tf_pwa.config_loader.extra
 - module, 87
- tf_pwa.config_loader.multi_config
 - module, 87
- tf_pwa.config_loader.particle_function
 - module, 88
- tf_pwa.config_loader.plot
 - module, 88
- tf_pwa.config_loader.plotter
 - module, 90
- tf_pwa.config_loader.sample
 - module, 93
- tf_pwa.cov_ten_ir
 - module, 133
- tf_pwa.data
 - module, 133
- tf_pwa.data_trans
 - module, 94
- tf_pwa.data_trans.dalitz
 - module, 94
- tf_pwa.data_trans.helicity_angle
 - module, 94
- tf_pwa.dec_parser
 - module, 137
- tf_pwa.dfun
 - module, 138
- tf_pwa.einsum
 - module, 140
- tf_pwa.err_num
 - module, 140
- tf_pwa.experimental
 - module, 95
- tf_pwa.experimental.build_amp
 - module, 95
- tf_pwa.experimental.extra_amp
 - module, 95
- tf_pwa.experimental.extra_data
 - module, 95
- tf_pwa.experimental.extra_function
 - module, 96
- tf_pwa.experimental.factor_system
 - module, 96
- tf_pwa.experimental.opt_int
 - module, 97
- tf_pwa.experimental.wrap_function
 - module, 97
- tf_pwa.expermental
 - module, 141
- tf_pwa.fit
 - module, 141
- tf_pwa.fit_improve
 - module, 142
- tf_pwa.fitfractions
 - module, 144
- tf_pwa.formula
 - module, 145
- tf_pwa.function
 - module, 145
- tf_pwa.generator
 - module, 97
- tf_pwa.generator.breit_wigner
 - module, 97
- tf_pwa.generator.generator
 - module, 98
- tf_pwa.generator.interp_nd

module, 98
 tf_pwa.generator.linear_interpolation
 module, 99
 tf_pwa.generator.plane_2d
 module, 99
 tf_pwa.generator.square_dalitz_plot
 module, 100
 tf_pwa.gpu_info
 module, 145
 tf_pwa.histogram
 module, 145
 tf_pwa.main
 module, 147
 tf_pwa.model
 module, 102
 tf_pwa.model.cfit
 module, 102
 tf_pwa.model.custom
 module, 104
 tf_pwa.model.model
 module, 106
 tf_pwa.model.opt_int
 module, 115
 tf_pwa.params_trans
 module, 147
 tf_pwa.particle
 module, 147
 tf_pwa.phasespace
 module, 152
 tf_pwa.root_io
 module, 154
 tf_pwa.significance
 module, 154
 tf_pwa.tensorflow_wrapper
 module, 154
 tf_pwa.transform
 module, 155
 tf_pwa.utils
 module, 155
 tf_pwa.variable
 module, 157
 tf_pwa.version
 module, 165
 tf_pwa.vis
 module, 165
 tf_pwa.weight_smear
 module, 166
 time_print() (in module tf_pwa.utils), 157
 to_complex() (in module tf_pwa.breit_wigner), 127
 topology_id() (DecayChain method), 150
 topology_map() (DecayChain method), 150
 topology_same() (DecayChain method), 150
 topology_structure() (DecayGroup method), 151
 total_size() (PlotData method), 90

total_size() (PlotDataGroup method), 90
 touch_var() (in module
 tf_pwa.config_loader.data_root_lhcb), 86
 trainable_variables (AbsPDF property), 44
 trainable_variables (BaseModel property), 107
 trainable_variables (VarsManager property), 164
 trans() (ParamsTrans method), 147
 trans_error_matrix() (VarsManager method), 164
 trans_f_grad_hess() (VarsManager method), 164
 trans_fcn_grad() (VarsManager method), 164
 trans_grad_hessp() (VarsManager method), 164
 trans_model() (in module tf_pwa.amp.core), 63
 trans_node_order() (in module
 tf_pwa.config_loader.sample), 94
 trans_params() (VarsManager method), 164
 TriangleGenerator (class in
 tf_pwa.generator.plane_2d), 99
 tuple_table() (in module tf_pwa.utils), 157
 twoBodyCMmom() (in module tf_pwa.breit_wigner), 127

U

uniform() (in module tf_pwa.histogram), 146
 UniformGenerator (class in tf_pwa.phasespace), 153
 unit() (Vector3 method), 121
 update() (GaussianConstr method), 110
 using_amplitude() (in module tf_pwa.config), 132

V

validate_file_name() (in module
 tf_pwa.config_loader.config_loader), 85
 value (NumberError property), 140
 value (Variable property), 160
 value_and_grad() (BaseCustomModel method), 105
 value_and_grad() (in module tf_pwa.amp.core), 63
 Variable (class in tf_pwa.variable), 158
 variable_scope() (in module tf_pwa.amp.core), 63
 variables (AbsPDF property), 44
 variables (Variable property), 160
 VarsManager (class in tf_pwa.variable), 161
 vect() (LorentzVector method), 120
 Vector3 (class in tf_pwa.angle), 120

W

wave_function() (in module tf_pwa.amp.cov_ten), 64
 weighted_kde() (in module tf_pwa.histogram), 146
 WeightedData (class in tf_pwa.histogram), 146
 WFunc1() (in module tf_pwa.cov_ten_ir), 133
 WFunc2() (in module tf_pwa.cov_ten_ir), 133
 wignerDx() (in module tf_pwa.cov_ten_ir), 133
 WrapFun (class in tf_pwa.experimental.wrap_function),
 97

X

xy2rp() (VarsManager method), 164

`xy2rp_all()` (*VarsManager* method), [165](#)
`xyzToangle()` (*in module tf_pwa.cov_ten_ir*), [133](#)