1 What do we want to do

A PWA model is described by a model function f_{model}, which often has the form:

$$f_{model}(y, \theta) = |\theta_1 * A_1(y) + \theta_2 * A_2(y) + ...|^2. \tag{1}$$

where:

$$y = (m^2_{ab}, m^2_{bc}, ...)$$

is the set of variables describing an event. We call its length `NUM_VAR` or `num_variables()`. Further,

$$\theta = (\theta_1, \theta_2, ...)$$

is the set of parameters we intend to fit. Its length is called `NUM_RES` or `num_resonances()`, since in the case described above (1) it coincides with the number of PWA resonances A_i.

The intention of this module is

a) to make a STAN model that takes parameter θ and then generates events $y_i = (m^2_{ab,i}, m^2_{bc,i}, ...)$; i.e., we sample y over $f_{model}(y, \theta)$;

b) to make a STAN model that takes many events y_i as data and then fits the parameter θ to the data; i.e., we sample θ over $f_{model}(y, theta)/Norm(theta)$.

The function

$$Norm(\theta) = \int f_{model}(y, \theta)\, dy$$

is the function that normalizes the PWA model function f_{model} for different parameter values θ.

2 How the model is implemented

2.1 The model

The functions $f_{model}, A_{cv} = (A_1, A_2, \ldots)$, $Norm()$ and the constants `NUM_RES` and `NUM_VAR` are specified in the file `'lib/c_lib/model.hpp'`. You must adjust these functions and constants to your PWA model. (If your model is of the type (1), you only need to adjust A_{cv} and the constants. The function $A_{cv}()$ returns a complex vector; its elements are the PWA amplitudes.)

2.2 Complex numbers

Stan does not support complex numbers in its language. To work around that, we describe complex numbers as 2d arrays of real numbers; complex vectors as 2d arrays of real vectors, etc. Some common complex number operations are provided in the folder `'lib/c_lib/complex/'`.

2.3 PWA library

There are some functions that often come up in PWA, such as Breit-Wigner or Flatte form factors, Zemach tensors, etc. Some of these functions are implemented in `'lib/c_lib/fct'`.

Sometimes it is useful to pack several constants (like masses or radii of the particles) into a structure (called particles) that can be used in code. Such objects are stored in `'lib/c_lib/structures'`.

2.4 Python wrapper

C++ is, of course, a great language, but I just like to use python for some on-the-go visualizing of the programmed functions. The folder `'lib/c_lib/py_wrapper'` contains a wrapper for the function `A_cv`. Call `python setup.py build` from this folder to assemble a python module 'lib/c_lib/py_wrapper/build/lib*/model.so'. To use the function from python, import this module and call something like

```
>>> import sys
>>> sys.path.insert(1,"<rel_path_to_meson_deca>/lib/c_lib/py_wrapper/build/lib*")
>>> import model
model.A_cv(model.num_variables(), y).
```

You can convert this output to a user-friendlier form using module `lib/py_lib/convert.py` by calling:

```
>>> sys.path.insert(1,"<rel_path_to_meson_deca>/lib/py_lib")
>>> import convert
>>> convert.ComplexVectorForm(model.A_cv(model.num_variables(), y)).
```

3 How to set up STAN models

3.1 Setting up

After we have adjusted the constants `NUM_RES`, `NUM_VAR`, and the function `A_cv` (and, if necessary, `f_model` and `Norm`), we want to assemble two STAN models to generate the data and to fit the data, respectively. The STAN files suited for such fitting are stored in `lib/stan_lib`, called `STAN_data_generator.stan` and `STAN_amplitude_fitting.stan`. Call the script
`...$ ./initialize_model.sh FOLDER_NAME`
to copy these files into `models/FOLDER_NAME`. The python wrapped `module.so` and a backup of `model.hpp` are copied into that folder as well. After that, it is necessary to adjust the files:

a) STAN_data_generator.data.R - you MUST adjust the parameter θ which is used for generation of events y_i.

b) STAN_data_generator.stan - you may adjust the boundary values of y_i here (not necessary, but recommended).

c) STAN_amplitude_fitting.stan - you MUST adjust which parameter θ_i is fixed (as the reference parameter) and which parameters are free. You may also adjust the boundary values of θ here.

3.2 Specifics of the assumption (1)

In the case when the function f_{model} has the form specified in (1), the function $Norm$ can be rewritten in the following convenient form:

$$Norm(\theta) = \int f_{model}(y, \theta)\, dy = \int | \sum_{r=1}^{NUM_RES} \theta_r A_{cv,r}(y)|^2 dy \tag{2}$$

$$= \int \sum_{r,r'=1}^{NUM_RES} \theta_r A_{cv,r}(y) A^*_{cv,r'}(y) \theta^*_{r'} dy \tag{3}$$

$$= \sum_{r,r'=1}^{NUM_RES} \theta_r \theta^*_{r'} \underbrace{\int A_{cv,r}(y) A^*_{cv,r'}(y)\, dy}_{=:I_{rr'}} . \tag{4}$$

With other words, the function $Norm(\theta)$ can be reformulated as a function $Norm(\theta, I_{rr'})$ with a pre-calculated matrix $I \in \mathbb{R}^{\texttt{NUM\_RES}\times\texttt{NUM\_RES}}$.

This is precisely how the function $Norm$ is implemented in `model.hpp` by default. After the model is set up in a folder as described above in Subsection 3.1, it is necessary to call
`../meson_deca/models/FOLDER_NAME$ ./../../utils/calculate_normalization_intergal.py y1_min y1_max y2_min y2_max <etc...>` This script, stored in `utils`, must be called from the model folder `FOLDER_NAME` that contains a python module `model.so`. This script calculates the matrix I and stores it in the file `normalization_integral.py`; the integrals in (4) are taken from `y1_min` to `y1_max`, from `y2_min` to `y2_max`, etc., respectively.

4 How to sample using STAN models

To build the STAN models into executables, call
`../cmdstan-2.6.2$ make meson_deca/models/FOLDER_NAME/STAN_data_generator`
`../cmdstan-2.6.2$ make meson_deca/models/FOLDER_NAME/STAN_amplitude_fitting`
from the CmdStan folder, or, alternatively, call
`../meson_deca/models/FOLDER_NAME$ ./../../build.sh` (you may also call ./../../clean.sh in case you need to delete the executables STAN_data_generator and STAN_amplitude_fitting).

After that, you may call the scripts
`../meson_deca/models/FOLDER_NAME$ ./../../generate.sh 10000` `../meson_deca/models/FOLDER`
`./../../fit.sh 1000` to generate 10000 events y_i and then to sample the distribution of the parameter θ 1000 times. If everything works correctly, the posterior distribution of θ should have peaks at the values specified in $STAN_data_generator.data.R$.

After you have made sure that everything works, you can fit θ for some real events y_i. Assume that the `.root` file with your events is stored in `$FOLDER/my_data.root`, and the trees containing data are called 'y.1', 'y.2', etc.

Call `utils/data_analysis__root_to_dataR.py my_data.root STAN_amplitude_fitting.data.R` to convert the events y_i to $A_{cv}(y_i)$ and to save them as STAN data file. After that, use `STAN_amplitude_fitting sample data file=STAN_amplitude_fitting.data.R` to generate the samples.