The ComPWA project

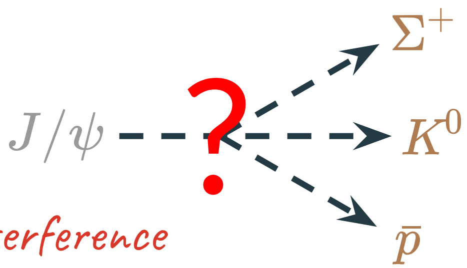
Speeding up amplitude analysis with a Computer Algebra System

Miriam Fritsch, **Remco de Boer**,
Wolfgang Gradl (JGU), **Stefan Pflüger**
Ruhr University Bochum

11 May 2023
CHEP2023

Amplitude analysis software

Aim: study of intermediate hadronic states



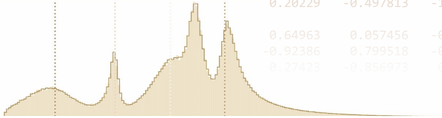
*Find **models that correctly** describe the observed **intensity distributions***

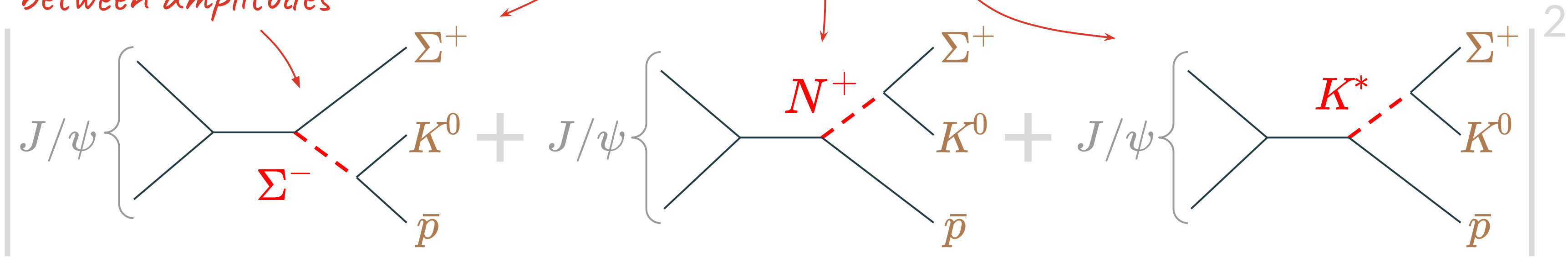
Interference between amplitudes

Input data

3 four-momenta per collision

E	p_x	p_y	p_z
0.05325	-0.102226	-0.271504	0.29496
1.30563	-0.324557	0.223228	1.37042
-1.35888	0.426783	0.048276	1.43152
-0.23327	0.509333	0.499320	0.75044
-0.68438	-0.801269	0.281889	1.09914
0.91766	0.291936	-0.781209	1.24733
-0.30031	0.284337	-0.255063	0.48589
-1.02024	-0.026281	0.630984	1.20746
1.32055	-0.258056	-0.375920	1.40356
0.55522	0.0865535	0.825067	0.99824
-0.75750	0.411259	0.234126	0.90331
0.20229	-0.497813	-1.059190	1.19534
0.64963	0.057456	-0.008806	0.65223
-0.92386	0.799518	-0.581799	1.35995
0.27423	-0.856973	0.590605	1.08473

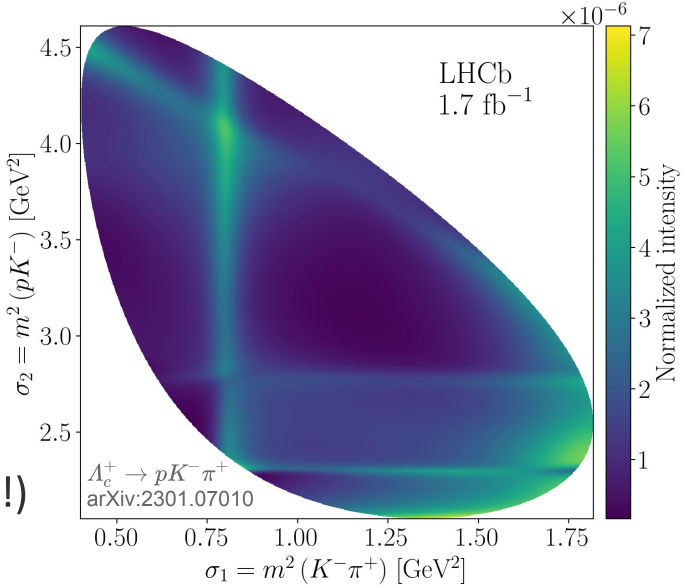




Amplitude analysis software

What makes it so difficult?

- Unbinned, multidimensional problem set
- Complicated parametrizations and estimators
  - need to quickly try out different parameterizations
  - fits can take several weeks
- Theory is hard to get into
- Relatively small community (but growing interest!)



Amplitude analysis software

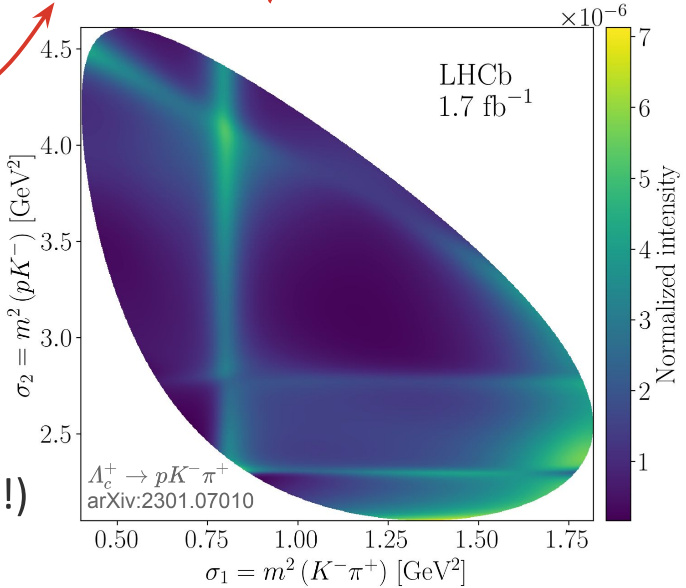
What makes it so difficult?

- Unbinned, multidimensional problem set
- Complicated parametrizations and estimators
  - need to quickly try out different parameterizations
  - fits can take several weeks
- Theory is hard to get into
- Relatively small community (but growing interest!)

fast computations

flexibility

documentation



pwa.rtfd.io

Amplitude analysis software

Has led to a large number of analysis packages and scripts

PWA frameworks

GPUPWA

TFPWA

Pawian

Laura++

RooFit

PyPWA

TensorFlowAnalysis

HYDRA
Multithreaded Data
Analysis Framework

AMPGEN

```
   ^
  / \
 /---\
/     \ MP
-------
   |
   | OOLS
```

GooFit
CUDA/OpenMP
Fitting Framework
for C++ & Python

pyhf
∂ifferentiable
ℒikelihoods

zfit

Scripts using fitter packages

pwa.rtfd.io

Amplitude analysis software

PWA frameworks

Has led to a large number of analysis packages and scripts

Trend: many frameworks try to become more modular

- Designed as a library
- Python/Julia bindings
- Flexibility through scripts instead of config files

→ Results in a more **flexible workflow** that can easily integrate new theories

scripts using fitter packages

Differentiable programming

Additional trend: several specialised packages from the ML and data science communities

NumPy

Numba

e.g. gradient descent algorithm

Not just Machine Learning!

Can be used for any fast numerical computations

Differentiable programming

Some of the techniques these back-ends offer:

- Vectorization
- Just-in-time compilation
- XLA (Accelerated Linear Algebra)
- Automatic differentiation
- Support for multithreading, GPUs, …

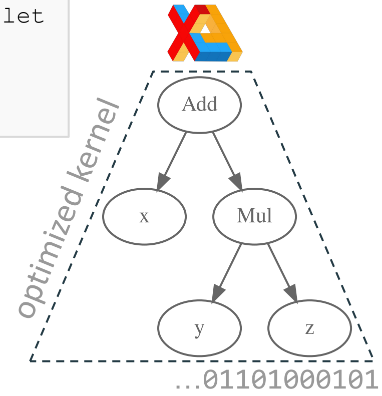
```python
@tf.function(jit_compile=True)
def my_expression(x, y, z):
    return x + y * z
```

Converted to device-agnostic XLA code

```
{ lambda ; a:i32[] b:i32[] c:i32[]. let
    d:i32[] = mul b c
    e:i32[] = add a d
  in (e,) }
```

Heavy lifting by optimized backend

```c
for (i = 0; i < rows; i++): {
  for (j = 0; j < columns; j++): {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```



Differentiable programming

Some of the techniques these back-ends offer:

- Vectorization
- Just-in-time compilation
- XLA (Accelerated Linear Algebra)
- Automatic differentiation
- Support for multithreading, GPUs, …

```
@tf.function(jit_compile=True)
def my_expression(x, y, z):
    return x + y * z
```

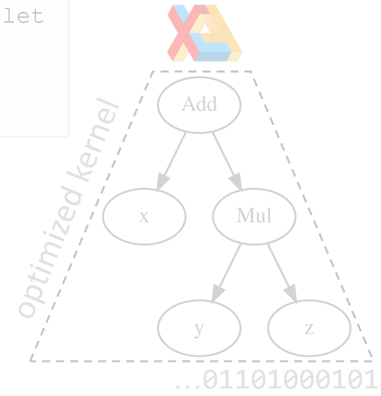
Usually all that the user needs to do

Converted to device-agnostic XLA code

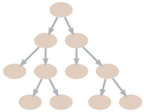
```
{ lambda ; a:i32[] b:i32[] c:i32[]. let
    d:i32[] = mul b c
    e:i32[] = add a d
  in (e,) }
```

Heavy lifting by optimized backend

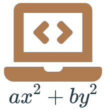
```
for (i = 0; i < rows; i++): {
  for (j = 0; j < columns; j++): {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```



How to bring code closer to theory?

✔ 

High performance through **computational back-ends** from ML and data science



Flexibility through a **Computer Algebra System**

Academic continuity through **living documentation**

Symbolic amplitude models

A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

```python
import sympy as sp
N, s, m0, w0 = sp.symbols("N s m0 Gamma0")
N / (m0**2 - sp.I * m0 * w0 - s)
```

SymPy

$$\frac{N}{m_0^2 - i m_0 \Gamma_0 - s}$$

Quite common already for theoreticians: quickly inspect and visualize some lineshape with Maple, Mathematica, Matlab, etc…

Symbolic amplitude models

A new technique: formulate your amplitude model with a Computer Algebra System

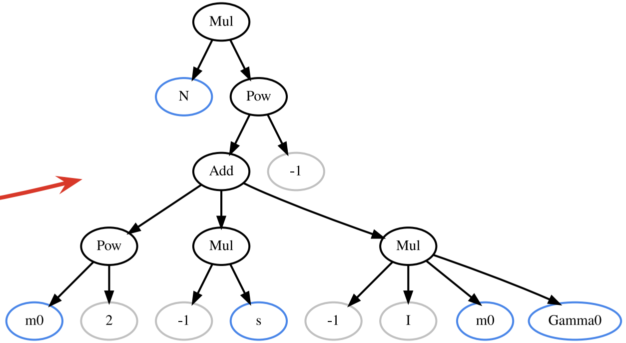
- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

```python
import sympy as sp
N, s, m0, w0 = sp.symbols("N s m0 Gamma0")
N / (m0**2 - sp.I * m0 * w0 - s)
```

SymPy

$$\frac{N}{m_0^2 - i m_0 \Gamma_0 - s}$$

CAS represents expression as a tree

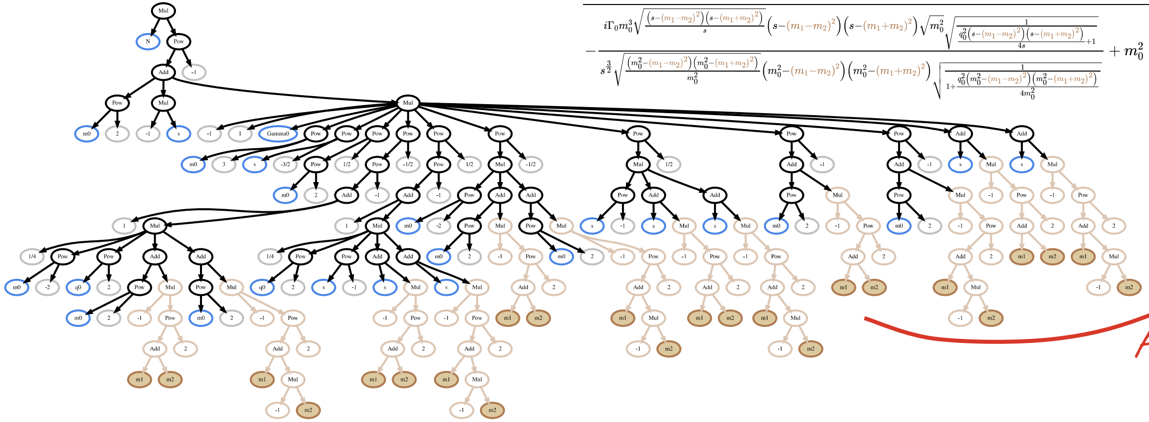


Symbolic amplitude models

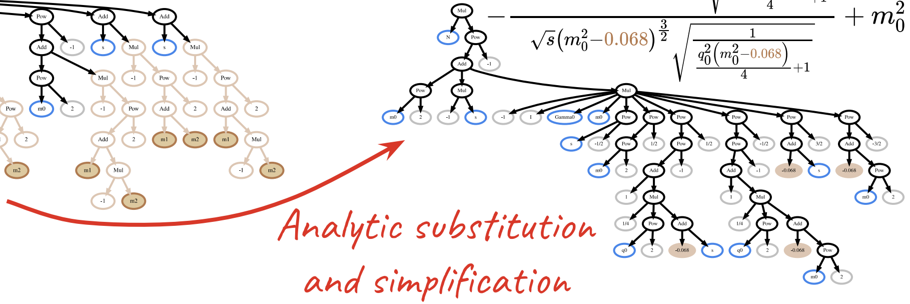
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

$$\frac{N}{-\frac{i\Gamma_0 m_0^3\sqrt{\frac{\left(s-(m_1-m_2)^2\right)\left(s-(m_1+m_2)^2\right)}{s}}\left(s-(m_1-m_2)^2\right)\left(s-(m_1+m_2)^2\right)\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2\left(s-(m_1-m_2)^2\right)\left(s-(m_1+m_2)^2\right)}{4s}+1}}}{s^{\frac{3}{2}}\sqrt{\frac{\left(m_0^2-(m_1-m_2)^2\right)\left(m_0^2-(m_1+m_2)^2\right)}{m_0^2}}\left(m_0^2-(m_1-m_2)^2\right)\left(m_0^2-(m_1+m_2)^2\right)\sqrt{\frac{1}{1+\frac{q_0^2\left(m_0^2-(m_1-m_2)^2\right)\left(m_0^2-(m_1+m_2)^2\right)}{4m_0^2}}}}+m_0^2-s}$$



$$\frac{N}{-\frac{i\Gamma_0 m_0(s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$



Symbolic amplitude models

$ax^2 + by^2$

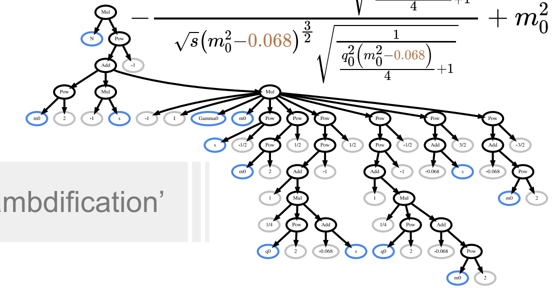
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

```
function out1 = my_expr(Gamma0, N, m0, s)

 out1 = N./(-1i*Gamma0.*m0.^3.*sqrt((s - 0.25).*(s -
0.01)./s).*(1 + (m0.^2 - 0.25).*(m0.^2 - 0.01)./(4*m0.^2)).*(s
- 0.25).*(s - 0.01).*sqrt(m0.^2)./(s.^(3/2).*sqrt((m0.^2 -
0.25).*(m0.^2 - 0.01)./m0.^2).*(1 + (s - 0.25).*(s -
0.01)./(4*s)).*(m0.^2 - 0.25).*(m0.^2 - 0.01)) + m0.^2 - s);

end
```

SymPy 'lambdification'

$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$



Symbolic amplitude models

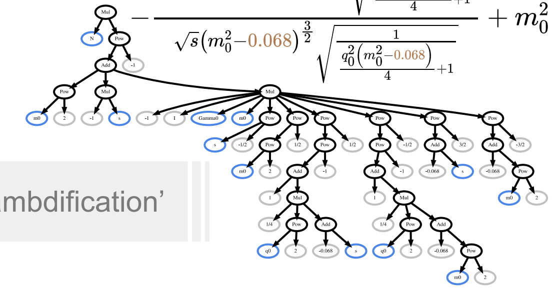
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

```fortran
REAL*8 function my_expr(Gamma0, N, m0, s)
implicit none
REAL*8, intent(in) :: Gamma0
REAL*8, intent(in) :: N
REAL*8, intent(in) :: m0
REAL*8, intent(in) :: s

my_expr = N/(-cmplx(0,1)*Gamma0*m0**3*sqrt((s - 0.25d0)*(s - 0.01d0)/s)* & (1 +
      (1.0d0/4.0d0)*(m0**2 - 0.25d0)*(m0**2 - 0.01d0)/m0**2)*(s - & 0.25d0)*(s -
      0.01d0)*sqrt(m0**2)/(s**(3.0d0/2.0d0)*sqrt((m0**2 - & 0.25d0)*(m0**2 -
      0.01d0)/m0**2)*(1 + (1.0d0/4.0d0)*(s - 0.25d0)*( & s - 0.01d0)/s)*(m0**2 -
      0.25d0)*(m0**2 - 0.01d0)) + m0**2 - s)
end function
```

SymPy ‘lambdification’

$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$



Symbolic amplitude models

$ax^2 + by^2$

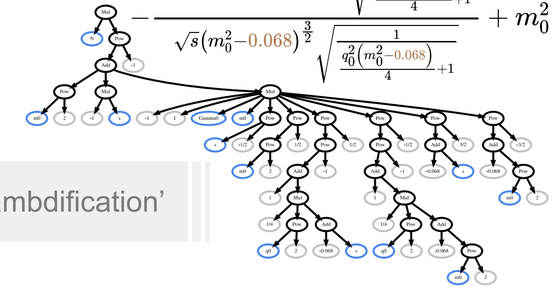
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

```c
// my_expr.h
#ifndef PROJECT__MY_EXPR__H
#define PROJECT__MY_EXPR__H
double my_expr(double Gamma0, double N, double m0, double s);
#endif

// my_expr.c
#include "my_expr.h"
#include <math.h>

double my_expr(double Gamma0, double N, double m0, double s) {
  double my_expr_result;
  return N/(-I*Gamma0*pow(m0, 3)*sqrt((s - 0.25)*(s - 0.01)/s)*(1 + (1.0/4.0)*(pow(m0, 2) -
    0.25)*(pow(m0, 2) - 0.01)/pow(m0, 2))*(s - 0.25)*(s - 0.01)*sqrt(pow(m0, 2))/(pow(s,
    3.0/2.0)*sqrt((pow(m0, 2) - 0.25)*(pow(m0, 2) - 0.01)/pow(m0, 2))*(1 + (1.0/4.0)*(s -
    0.25)*(s - 0.01)/s)*(pow(m0, 2) - 0.25)*(pow(m0, 2) - 0.01)) + pow(m0, 2) - s);
```

SymPy 'lambdification'

$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}} + m_0^2 - s}$$



Symbolic amplitude models

$ax^2 + by^2$

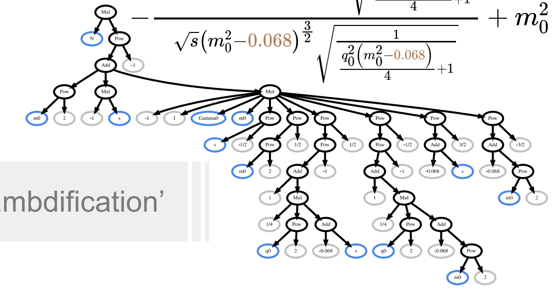
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications

```python
@jax.jit
def _lambdifygenerated(Gamma0, N, m0, s):
    return N / (
        -1j
        * Gamma0
        * m0
        * ((1 / 4) * m0**2 + 0.9831)
        * (s - 0.0676) ** (3 / 2)
        * sqrt(m0**2)
        / (sqrt(s) * (m0**2 - 0.0676) ** (3 / 2) * ((1 / 4) * s + 0.9831))
        + m0**2
        - s
    )
```

SymPy 'lambdification'

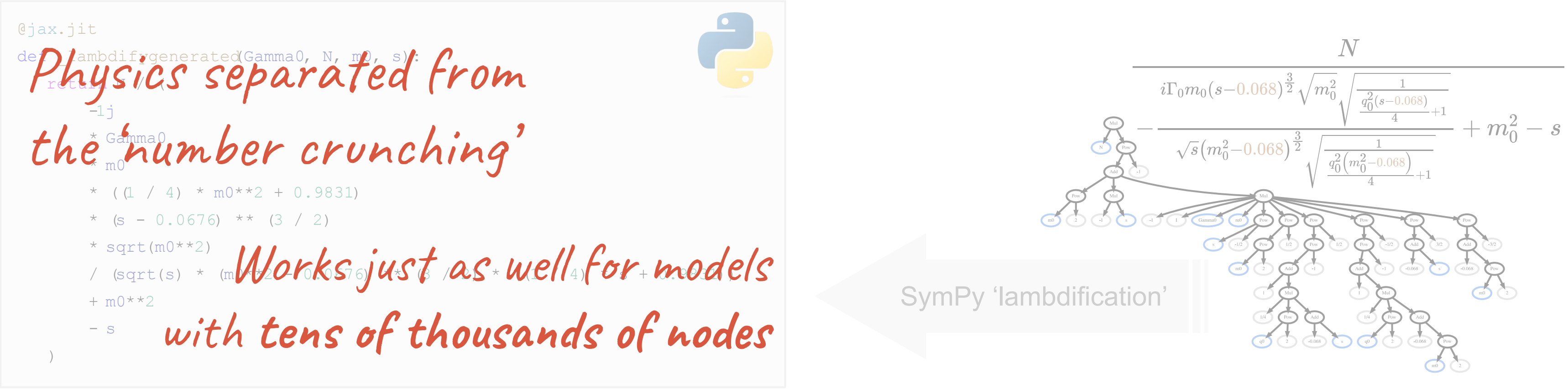
$$\frac{N}{- \frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}} \sqrt{m_0^2} \sqrt{\frac{1}{\frac{q_0^2 (s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}} \sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}} + m_0^2 - s}$$



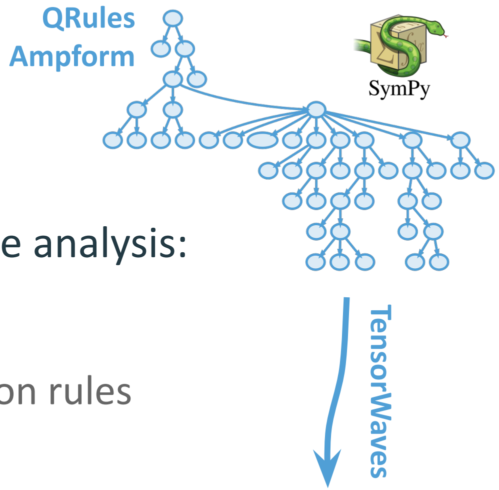
Symbolic amplitude models

A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Code generation: symbolic model as template to computational back-ends (SSoT)
- Improve computational performance with algebraic simplifications



The ComPWA project



Common Partial Wave Analysis

Three main Python packages that together cover a full amplitude analysis:

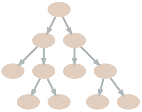
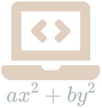
Physics
- **QRules** — Automated quantum number conservation rules
- **AmpForm** — Formulate symbolic amplitude models

Computations
- **TensorWaves** — Fit models to data and generate data samples with multiple computational back-ends

All are designed as **libraries**, so they can be used by other packages
by installing through pip or Conda

Demo in backup slides

How to bring code closer to theory?

- ✔  High performance through **computational back-ends** from ML and data science
- ✔ 

Flexibility through a **Computer Algebra System**
-  Academic continuity through **living documentation**

Living documentation

Codebase

```python
@implement_doit_method
class EnergyDependentWidth(UnevaluatedExpression):
    r"""Mass-dependent width, coupled to the pole position of the resonance.

    See :pdg-review:`2020; Resonances; p.6` and
    :cite:`asnerDalitzPlotAnalysis2006`, equation (6). Default value for
    :code:`phsp_factor` is :meth:`PhaseSpaceFactor`.

    Note that the `.BlattWeisskopfSquared` of AmpForm is normalized in the
    sense that equal powers of :math:`z` appear in the nominator and the
    denominator, while the definition in the PDG (as well as some other
    sources), always have :math:`1` in the nominator of the Blatt-Weisskopf. In
    that case, one needs an additional factor :math:`\left(q/q_0\right)^{2L}`
    in the definition for :math:`\Gamma(m)`.
    """

    def evaluate(self) -> sp.Expr:
        s, mass0, gamma0, m_a, m_b, angular_momentum, meson_radius = self.args
        q_squared = BreakupMomentumSquared(s, m_a, m_b)
        q0_squared = BreakupMomentumSquared(mass0**2, m_a, m_b)
        form_factor_sq = BlattWeisskopfSquared(
            angular_momentum,
            z=q_squared * meson_radius**2,
        )
        form_factor0_sq = BlattWeisskopfSquared(
            angular_momentum,
            z=q0_squared * meson_radius**2,
        )
        rho = self.phsp_factor(s, m_a, m_b)
        rho0 = self.phsp_factor(mass0**2, m_a, m_b)
        return gamma0 * (form_factor_sq / form_factor0_sq) * (rho / rho0)

    def _latex(self, printer: LatexPrinter, *args) -> str:
        s, _, width, *_ = self.args
        s = printer._print(s)
        subscript = _indices_to_subscript(_determine_indices(width))
        name = Rf"\Gamma{subscript}" if self._name is None else self._name
        return Rf"{name}\left({s}\right)"
```

Search the docs ...

Launch interactive examples

Pole parametrization

After all these matrix definitions, the final challenge is to choose a correct parametrization for the elements of $\boldsymbol{K}$ and $\boldsymbol{P}$ that accurately describes the resonances we observe.[3] There are several choices, but a common one is the following summation over the **poles** R:[4]

$$K_{ij} = \sum_R \frac{g_{R,i}g_{R,j}}{m_R^2 - s} + c_{ij}$$
$$\hat{K}_{ij} = \sum_R \frac{g_{R,i}(s)g_{R,j}(s)}{(m_R^2 - s)\sqrt{\rho_i \rho_j}} + \hat{c}_{ij} \qquad (14)$$

with $c_{ij}, \hat{c}_{ij}$ some optional background characterization and $g_{R,i}$ the residue functions. The residue functions are then often expressed as:

$$(15)$$

[4] Eqs. (75-78)

Jupyter notebooks

- Dynamic code allows for interactivity
- Can be rendered as web pages
- Serves as integration test

with $\gamma_{R,i}$ some *real* constants and $\Gamma^0_{R,i}$ the **partial width** of each pole. In the Lorentz-invariant form, the fixed width Γ^0 is replaced by an "energy dependent" `CoupledWidth` $\Gamma(s)$.[5] The **width** for each pole can be computed as $\Gamma^0_R = \sum_i \Gamma^0_{R,i}$.

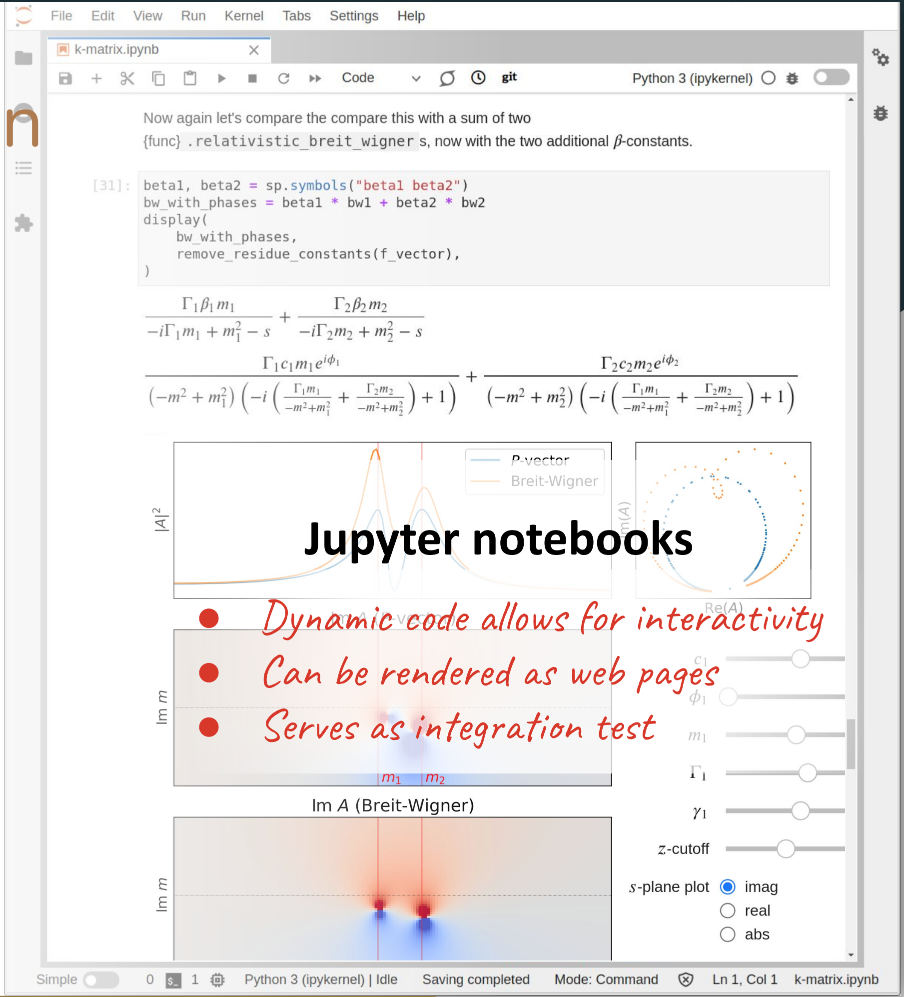
The production vector P is commonly parameterized

Living documentation

Codebase

```python
@implement_doit_method
class EnergyDependentWidth(UnevaluatedExpression):
    r"""Mass-dependent width, coupled to the pole position of the resonance.

    See :pdg-review:`2020; Resonances; p.6` and
    :cite:`asnerDalitzPlotAnalysis2006`, equation (6). Default value for
    :code:`phsp_factor` is :meth:`PhaseSpaceFactor`.

    Note that the `.BlattWeisskopfSquared` of AmpForm is normalized in the
    sense that equal powers of :math:`z` appear in the nominator and the
    denominator, while the definition in the PDG (as well as some other
    sources), always have :math:`1` in the nominator of the Blatt-Weisskopf. In
    that case, one needs an additional factor :math:`\left(q/q_0\right)^{2L}`
    in the definition for :math:`\Gamma(m)`.
    """

    def evaluate(self) -> sp.Expr:
        s, mass0, gamma0, m_a, m_b, angular_momentum, meson_radius = self.args
        q_squared = BreakupMomentumSquared(s, m_a, m_b)
        q0_squared = BreakupMomentumSquared(mass0**2, m_a, m_b)
        form_factor_sq = BlattWeisskopfSquared(
            angular_momentum,
            z=q_squared * meson_radius**2,
        )
        form_factor0_sq = BlattWeisskopfSquared(
            angular_momentum,
            z=q0_squared * meson_radius**2,
        )
        rho = self.phsp_factor(s, m_a, m_b)
        rho0 = self.phsp_factor(mass0**2, m_a, m_b)
        return gamma0 * (form_factor_sq / form_factor0_sq) * (rho / rho0)

    def _latex(self, printer: LatexPrinter, *args) -> str:
        s, _, width, *_ = self.args
        s = printer._print(s)
        subscript = _indices_to_subscript(_determine_indices(width))
        name = Rf"\Gamma{subscript}" if self._name is None else self._name
        return Rf"{name}\left({s}\right)"
```

Jupyter notebooks

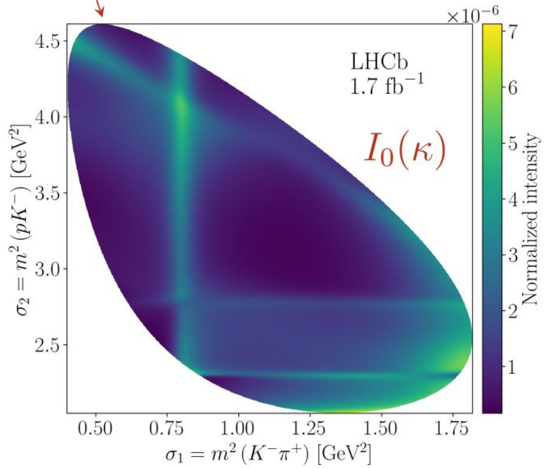


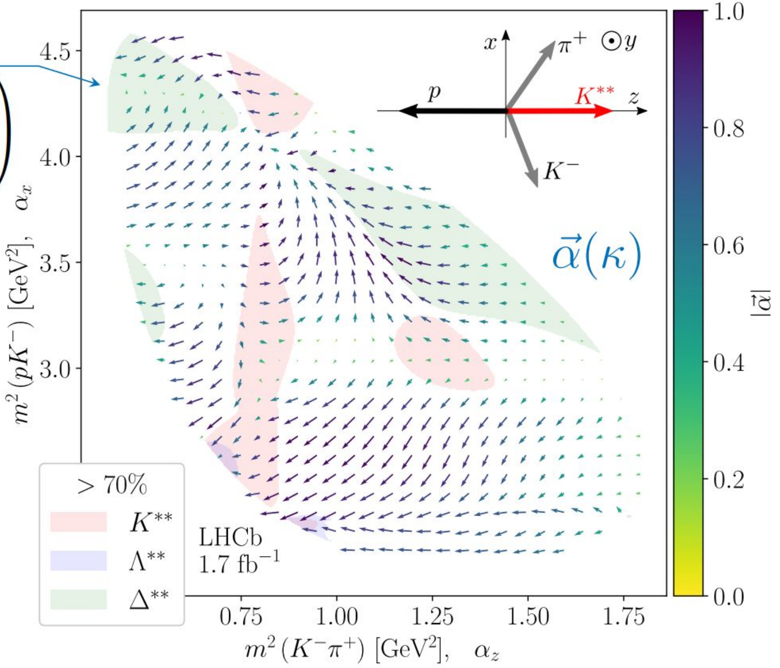
- Dynamic code allows for interactivity
- Can be rendered as web pages
- Serves as integration test

Living documentation

ComPWA recently enabled **vector field computations** for polarization studies

$$|\mathcal{M}(\phi,\theta,\chi,\kappa)|^2 = I_0(\kappa)\left(1 + \sum_{i,j=1}^{3} P_i R_{ij}(\phi,\theta,\chi)\alpha_j(\kappa)\right)$$





The self-documenting workflow allowed to publish the **full analysis as a website of notebooks**

Ongoing work and future ideas

- Main focus: Implement and test more symbolic spin formalisms and dynamics
  Dalitz-plot decomposition, K-matrix, spin density, tensor formalism…
- Improve integration into other HEP python packages
  e.g. standardise workflows, interfacing to zfit and scikit-hep package, … → PyHEP.dev
- Benchmark comparisons between amplitude analysis frameworks?
  Comparing workflows is hard and time-consuming, see e.g. this meeting

Ongoing work and future ideas

- Main focus: Implement and test more symbolic spin formalisms and dynamics
  Dalitz-plot decomposition, K-matrix, spin density, tensor formalism…
- Improve integration into other HEP python packages
  e.g. standardise workflows, interfacing to zfit and scikit-hep package, … → PyHEP.dev
- Benchmark comparisons between amplitude analysis frameworks?
  Comparing workflows is hard and time-consuming, see e.g. this meeting

Thank you for your attention!

Back-up

The main ComPWA packages

```
pip install qrules
pip install ampform
pip install tensorwaves
```

qrules.rtfd.io

QRules

Quantum number conservation rules

Core: 'search engine' for quantum numbers

Get particle properties:*

```
PDG = qrules.load_pdg()
PDG.find("a(2)(1320)0")
```

```
Particle(
  name='a(2)(1320)0',
  pid=115,
  latex='a_{2}(1320)^{0}',
  spin=2.0,
  mass=1.3182,
  width=0.107,
  isospin=Spin(1, 0),
  parity=+1,
  c_parity=+1,
  g_parity=-1,
)
```

Check which conservation rules are violated:

```
qrules.check_reaction_violations(
    initial_state="pi0",
    final_state=["gamma", "gamma", "gamma"],
)
```

launch binder | Open in Colab

```
{frozenset({'c_parity_conservation'})}
```

Also a library of conservation rules

Find particles by quantum number:

```
selection = PDG.filter(
    lambda p: p.mass > 2.8
    and p.spin > 0
    and p.charge
    and p.charmness
    and p.parity == +1
)
selection.names
```

```
['Lambda(c)(2880)+', 'Xi(c)(2815)~-']
```

* PDG info computed from the scikit-hep particle package

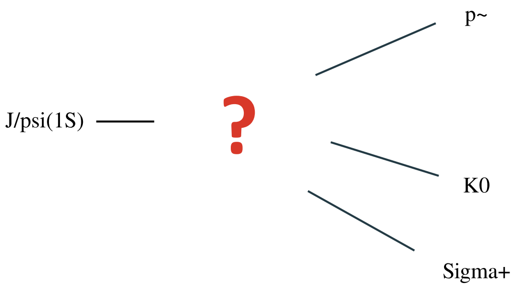
qrules.rtfd.io

QRules

Quantum number conservation rules

PWA use case: compute which particle reactions are allowed between a given initial and final state

1. User specifies some boundary conditions
   (particle names, allowed interactions, isobar model, etc.)

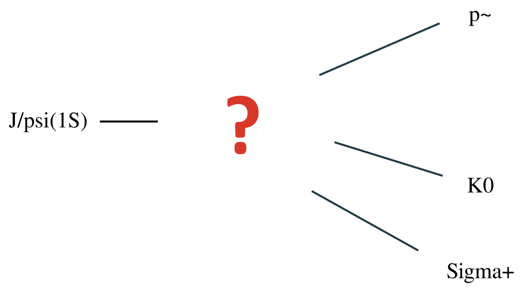


qrules.rtfd.io

QRules
Quantum number conservation rules

PWA use case: compute which particle reactions are allowed between a given initial and final state

1. User specifies some boundary conditions
   (particle names, allowed interactions, isobar model, etc.)
2. QRules then:
   - determines all possible decay topologies,



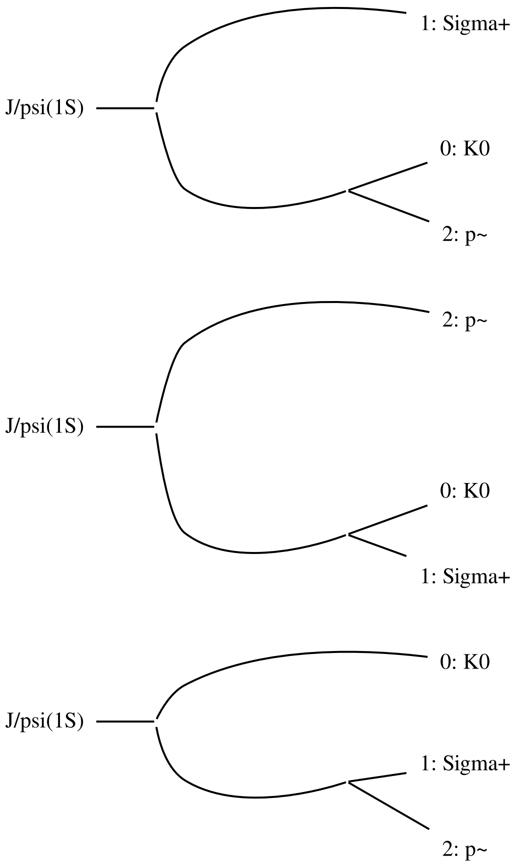
qrules.rtfd.io

QRules

Quantum number conservation rules

PWA use case: compute which particle reactions are allowed between a given initial and final state

1. User specifies some boundary conditions
   (particle names, allowed interactions, isobar model, etc.)
2. QRules then:
   - determines all possible decay topologies,
   - gets corresponding particle properties from the PDG
     (or any custom definitions),



qrules.rtfd.io

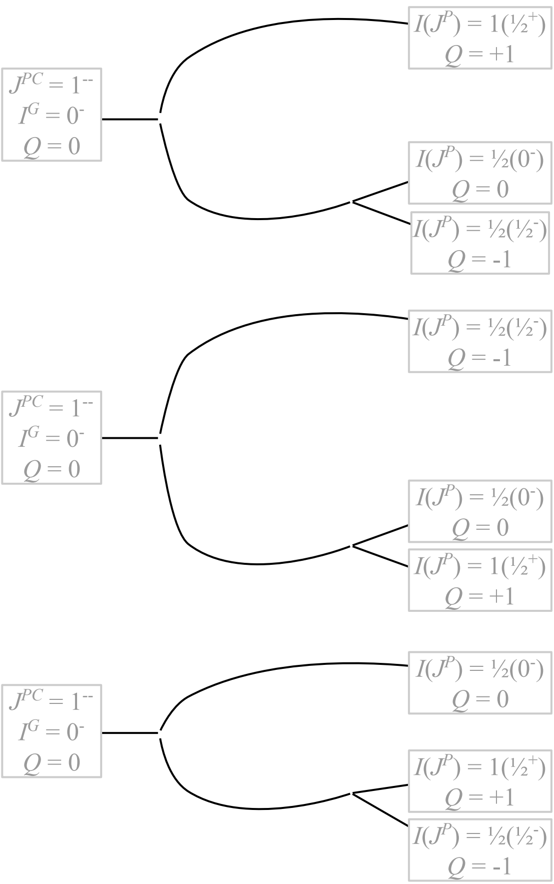
ComPWA

QRules

Quantum number conservation rules

PWA use case: compute which particle reactions are allowed between a given initial and final state

1. User specifies some boundary conditions
   (particle names, allowed interactions, isobar model, etc.)
2. QRules then:
   - determines all possible decay topologies,
   - gets corresponding particle properties from the PDG
     (or any custom definitions),



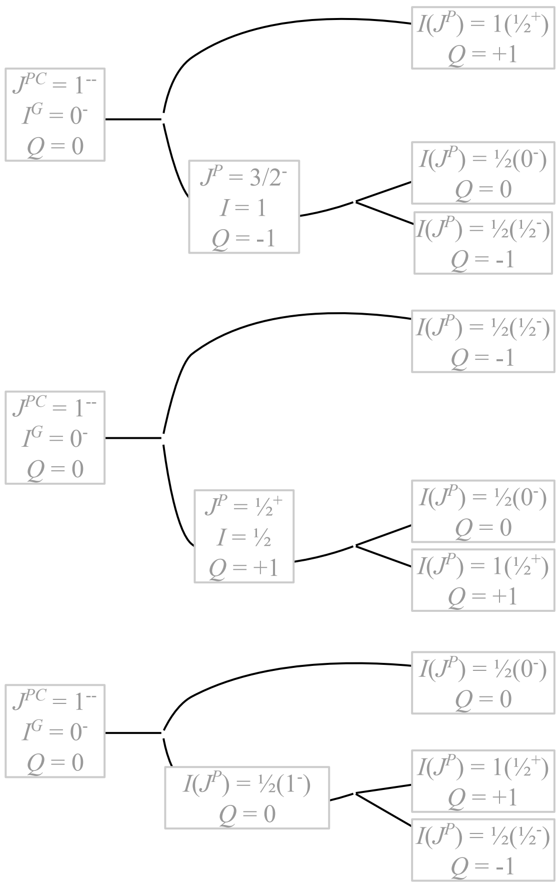
qrules.rtfd.io

QRules

Quantum number conservation rules

PWA use case: compute which particle reactions are allowed between a given initial and final state

1. User specifies some boundary conditions
   (particle names, allowed interactions, isobar model, etc.)
2. QRules then:
   - determines all possible decay topologies,
   - gets corresponding particle properties from the PDG
     (or any custom definitions),
   - propagates quantum numbers through intermediate edges,
   - and selects all allowed transitions with its conservation laws



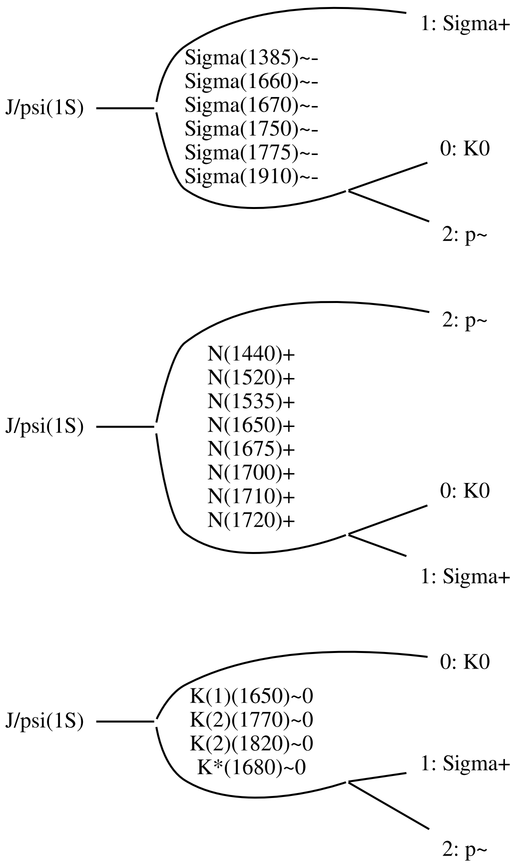
qrules.rtfd.io

QRules

Quantum number conservation rules

PWA use case: compute which particle reactions are allowed between a given initial and final state

1. User specifies some boundary conditions
   (particle names, allowed interactions, isobar model, etc.)
2. QRules then:
   - determines all possible decay topologies,
   - gets corresponding particle properties from the PDG
     (or any custom definitions),
   - propagates quantum numbers through intermediate edges,
   - and selects all allowed transitions with its conservation laws



qrules.rtfd.io

QRules

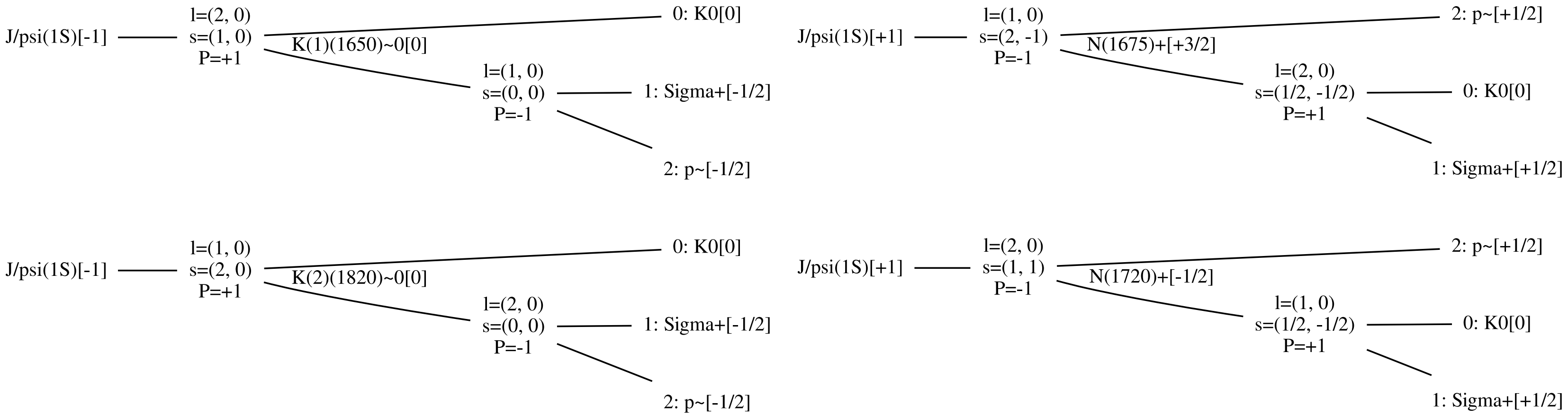
Quantum number conservation rules

The returned object contains **all information to formulate an amplitude model**!

```python
reaction = qrules.generate_transitions(
    initial_state="J/psi(1S)",
    final_state=["K0", "Sigma+", "p~"],
    allowed_interaction_types=["strong"],
)
```

Selects conservation rules

launch binder | Open in Colab



ampform.rtfd.io

AmpForm

Symbolic amplitude model formulation

- Library of **spin formalisms** and **dynamics**
- Formulate QRules' state transitions as an amplitude model
- Formulated as **algebraic expressions** (SymPy)
- Serves as **template to a computational back-end** for fitting and generating data distributions

```python
n = sp.Symbol("n_R")
matrix = RelativisticKMatrix.formulate(
    n_channels=1,
    n_poles=n,
)
matrix[0, 0]
```

$$\frac{\rho(s) \sum_{R=1}^{n_R} \frac{\Gamma(s)\gamma_{R,0}^2 m_R}{-s+m_R^2}}{-i\rho(s) \sum_{R=1}^{n_R} \frac{\Gamma(s)\gamma_{R,0}^2 m_R}{-s+m_R^2} + 1}$$

```python
matrix = NonRelativisticKMatrix.formulate(
    n_channels=2,
    n_poles=1,
).doit()
matrix[0, 0].simplify()
```

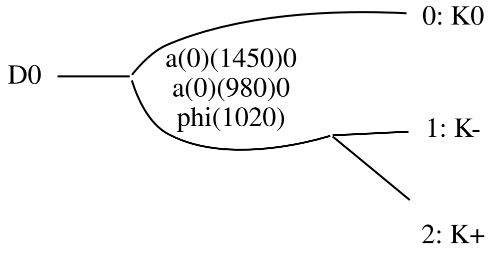
launch binder | Open in Colab

$$-\frac{\Gamma_{1,0}\gamma_{1,0}^2 m_1}{s + i\Gamma_{1,0}\gamma_{1,0}^2 m_1 + i\Gamma_{1,1}\gamma_{1,1}^2 m_1 - m_1^2}$$

ampform.rtfd.io

AmpForm

Symbolic amplitude model formulation



Example: amplitude model for $D^0 \to K^0\ K^-\ K^+$ with 3 resonances

```python
builder = ampform.get_builder(reaction)
resonances = reaction.get_intermediate_particles()
for p in resonances:
    builder.set_dynamics(p.name, create_relativistic_breit_wigner_with_ff)
builder.set_dynamics("a(0)(980)0", create_analytic_breit_wigner)
model = builder.formulate()
```

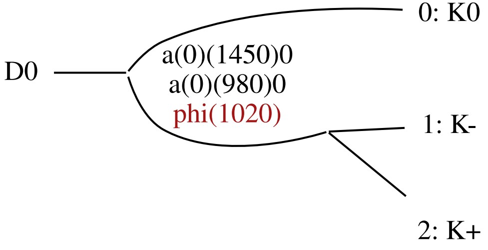
launch binder | Open in Colab

$$\left|A_{D^0_0\to K^0_0\phi(1020)_0;\phi(1020)_0\to K^+_0K^-_0} + A_{D^0_0\to K^0_0 a_0(1450)^0_0;a_0(1450)^0_0\to K^+_0K^-_0} + A_{D^0_0\to K^0_0 a_0(980)^0_0;a_0(980)^0_0\to K^+_0K^-_0}\right|^2$$

ampform.rtfd.io

AmpForm

Symbolic amplitude model formulation



Example: amplitude model for $D^0 \to K^0\ K^-\ K^+$ with 3 resonances

```python
builder = ampform.get_builder(reaction)
resonances = reaction.get_intermediate_particles()
for p in resonances:
    builder.set_dynamics(p.name, create_relativistic_breit_wigner_with_ff)
builder.set_dynamics("a(0)(980)0", create_analytic_breit_wigner)
model = builder.formulate()
```

launch binder | Open in Colab

$$\left|A_{D^{0}_{0}\to K^{0}_{0}\phi(1020)_{0};\phi(1020)_{0}\to K^{+}_{0}K^{-}_{0}} + A_{D^{0}_{0}\to K^{0}_{0}a_{0}(1450)^{0}_{0};a_{0}(1450)^{0}_{0}\to K^{+}_{0}K^{-}_{0}} + A_{D^{0}_{0}\to K^{0}_{0}a_{0}(980)^{0}_{0};a_{0}(980)^{0}_{0}\to K^{+}_{0}K^{-}_{0}}\right|^2$$

Each amplitude can be further inspected:

```python
model.components[R"A_{D^{0}_{0} \to K^{0}_{0} \phi(1020)_{0}; \phi(1020)_{0} \to K^{+}_{0} K^{-}_{0}}"]
```

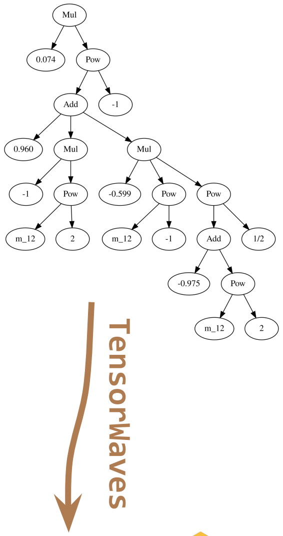
$$\frac{C_{D^0\to K^0_0\phi(1020)_0;\phi(1020)\to K^+_0K^-_0}\Gamma_{\phi(1020)}m_{\phi(1020)}\sqrt{B_1^2\left(d^2_{\phi(1020)}q^2_{122}\left(m^2_{12}\right)\right)}D^0_{0,0}(-\phi_0,\theta_0,0)D^1_{0,0}\left(-\phi_1^{12},\theta_1^{12},0\right)}{-m^2_{12}+m^2_{\phi(1020)}-im_{\phi(1020)}\Gamma_{1020}\left(m^2_{12}\right)}$$

tensorwaves.rtfd.io

ComPWA

TensorWaves

Fit and generate data with computational back-ends



TensorWaves responsibilities:

- Express mathematical expressions in a computational back-end
- Generate (deterministic) amplitude-based Monte Carlo samples
- Perform unbinned fits with different back-ends
  (TensorFlow, NumPy, JAX, …)
- Also integrates different optimizers (Minuit2, SciPy, …)

NumPy Numba

Any symbolic input

```
function = create_parametrized_function(expression, parameter_defaults, backend="jax")
estimator = UnbinnedNLL(function, data, phsp, backend="jax")
optimizer = Minuit2(callback=CSVSummary("fit_traceback.csv"))
fit_result = optimizer.optimize(estimator, initial_parameters)
```

Open in Colab

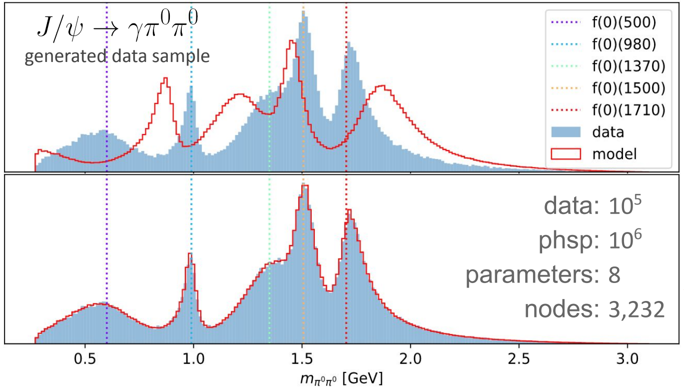
tensorwaves.rtfd.io

TensorWaves

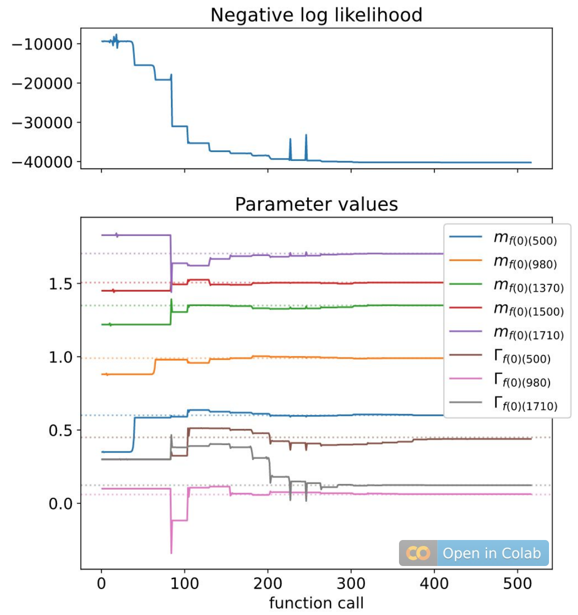
Fit and generate data with computational back-ends

Does it work? JAX+Minuit2 example benchmark:

Intel Core i7-8750H CPU @ 2.20GHz 12 cores	**56s**
GeForce GTX 1050 Mobile GPU @ 1.35GHz	**47s**
Tesla K80 GPU (Colab)	**15s**
Intel Xeon CPU @ 2.20GHz 1 core (Colab)	**3m20**



optimise parameters



tensorwaves.rtfd.io

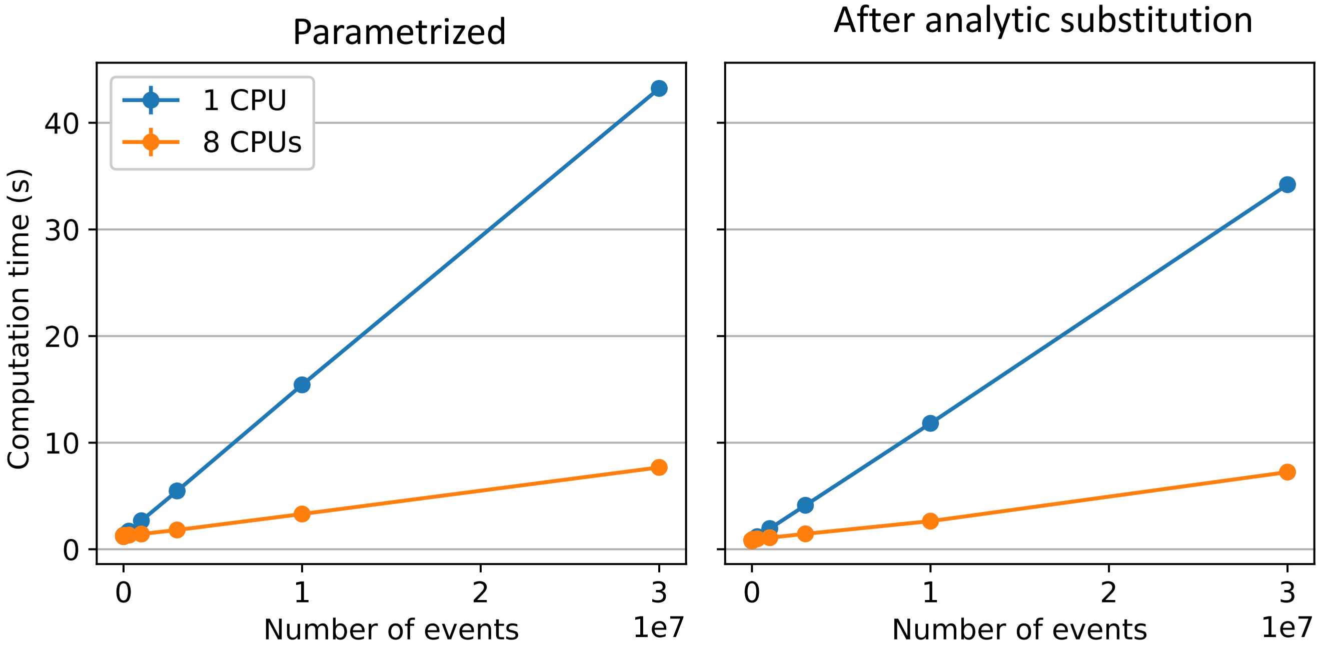
TensorWaves

Fit and generate data with computational back-ends

Amplitude model for $\Lambda_c \to p\pi K$
12 resonances, 59 parameters,
DPD alignment for 3 subsystems

Expression tree complexity:
parametrized: 43,198 nodes
substituted: 9,624 nodes

Backend: JAX
CPU: Intel i7-8750H 2.20GHz
$\to$ computation time **decreases by 25%**



tensorwaves.rtfd.io

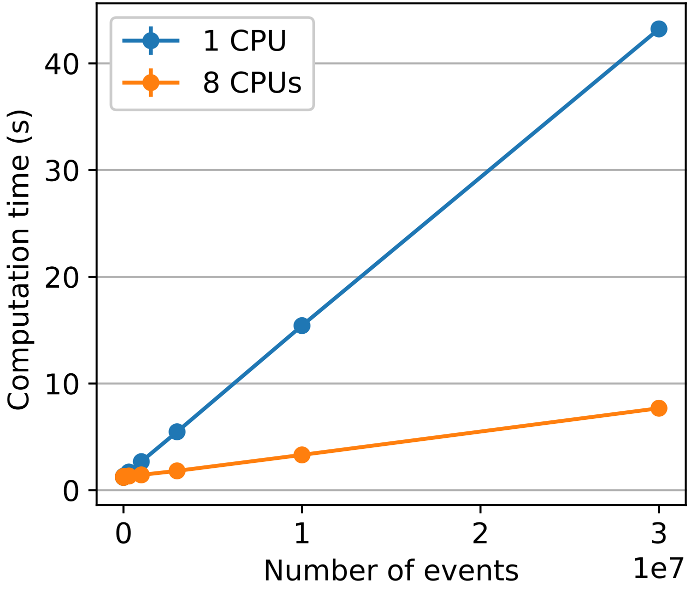
TensorWaves

Fit and generate data with computational back-ends

Amplitude model for $\Lambda_c \to p\pi K$
12 resonances, 59 parameters,
DPD alignment for 3 subsystems

Expression tree complexity:
  parametrized: 43,198 nodes
  substituted: 9,624 nodes

SymPy+JAX (Python)



Julia