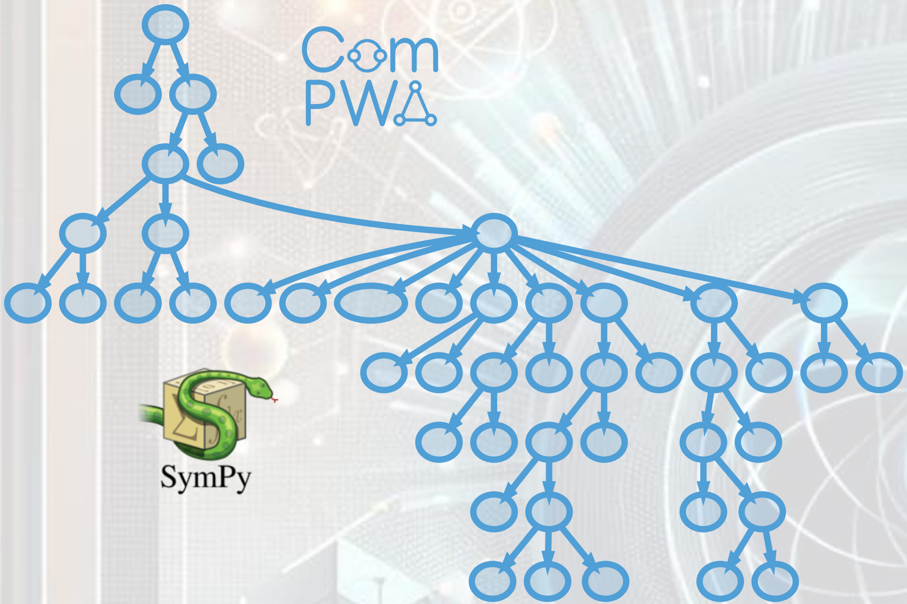
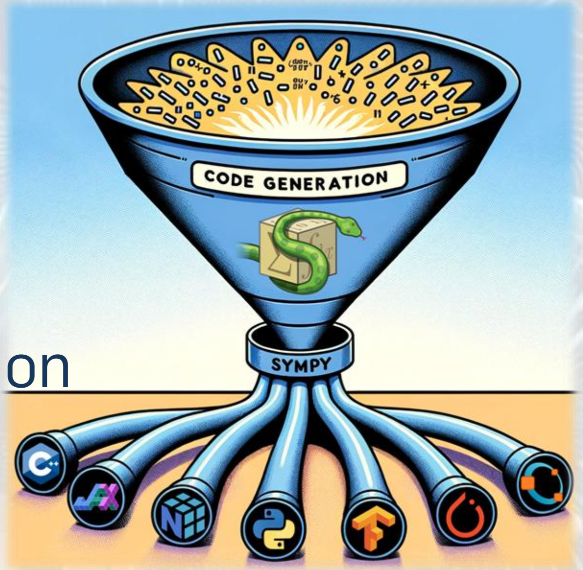


RUHR-UNIVERSITÄT BOCHUM

Symbolic computation and model preservation

Common Partial Wave Analysis (ComPWA)



December 14th, 2023
bit.ly/compwa-comap-2023

COMAP-V Workshop

Remco de Boer

Overview

- **Challenges for amplitude analysis software**
- **New technologies**
  - Outsourcing heavy computations to ML packages
  - Inserting a Computer Algebra System
  - Effect on documentation and preservation
- **Application example:** polarimeter vector field by LHCb
- **CAS models and serialization** ←
- Points for discussion

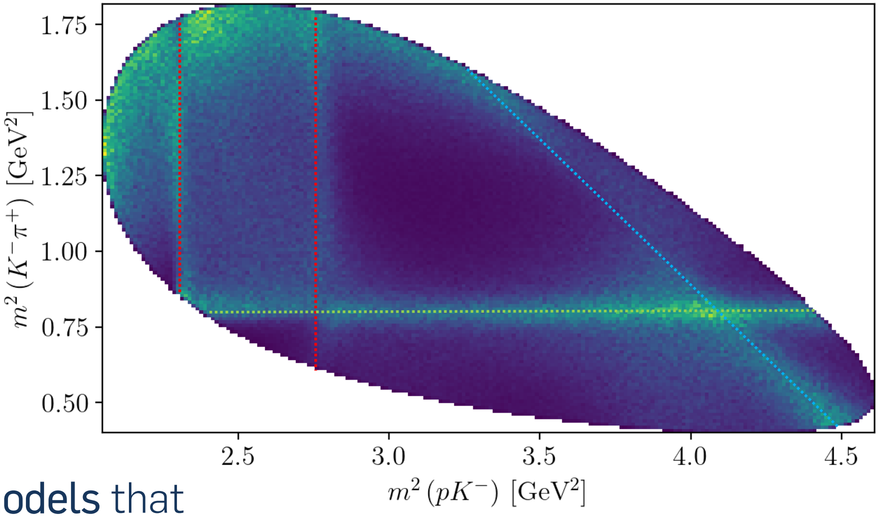
Amplitude analysis software

Input data

3 four-momenta per collision

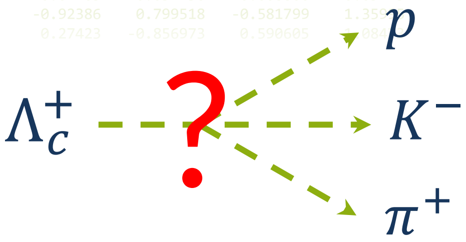
E	p_x	p_y	p_z
0.05325	-0.102226	-0.271504	0.29496
1.30563	-0.324557	0.223228	1.37042
-1.35888	0.426783	0.048276	1.43152
-0.23327	0.509333	0.499320	0.75044
-0.68438	-0.801269	0.281889	1.09914
0.91766	0.291936	-0.781209	1.24733
-0.30031	0.284337	-0.255063	0.48589
-1.02024	-0.026281	0.630984	1.20746
1.32055	-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	1.359
0.27423	-0.856973	0.590605	1.084

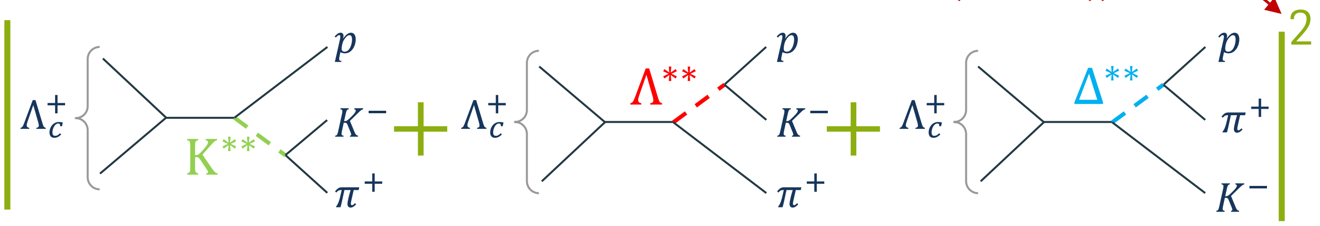
Our aim: investigate the spectrum to understand **intermediate hadronic states**



Method: find **amplitude models** that correctly describe the observed **intensity distributions**

*Complex-valued amplitudes describe **interference effects***





*Different **parametrizations** per amplitude*

Amplitude analysis software

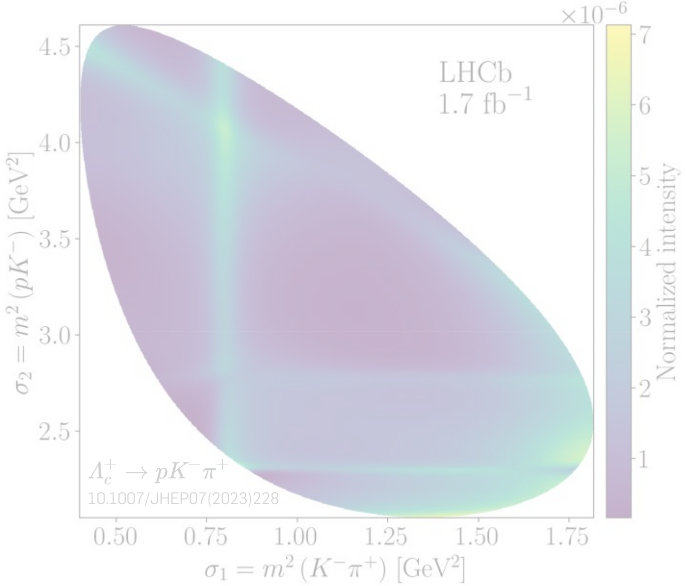
What makes PWA so challenging?

- Unbinned, multidimensional problem set
- Complicated parametrizations and estimators
  - need to quickly try out different models
  - each fit takes a long time
- Theory is hard to get into
- Relatively small community (but growing interest!)
- Hard to reproduce results (?)

performance

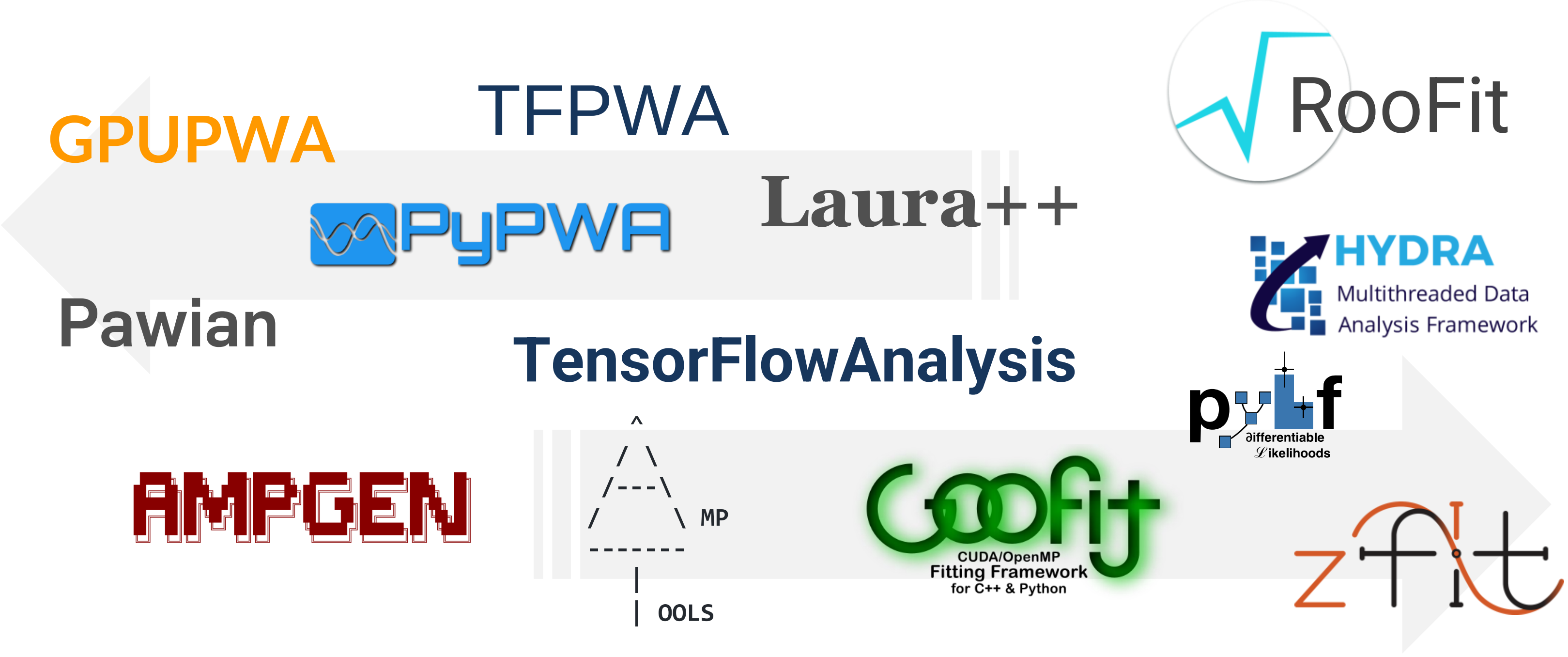
flexibility

documentation
and preservation



Amplitude analysis software

These unique challenges have led to a large number of analysis packages and scripts



PWA frameworks

GPUPWA

TFPWA

PyPWA

Laura++

Pawian

TensorFlowAnalysis

AMPGEN

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

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

RooFit

HYDRA
Multithreaded Data
Analysis Framework

pyhf
∂ifferentiable
ℒikelihoods

zfit

Scripts using fitter packages

Amplitude analysis software

These unique challenges have led to a large number of analysis packages and scripts

PWA frameworks

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 **dynamic and interactive workflow** that can easily integrate new theories

Scripts using fitter packages

Mission: bring code closer to theory

Outsource computations to **array-oriented backends** from the ML and data science community

Flexibility through a **Computer Algebra System**

$ax^2 + by^2$

Academic continuity through **living documentation**

Model preservation!

Array-oriented backends

Past few years saw the emergence of several specialised packages from the ML and data science communities

NumPy

JAX

PyTorch

Numba

TensorFlow

e.g. gradient descent algorithm

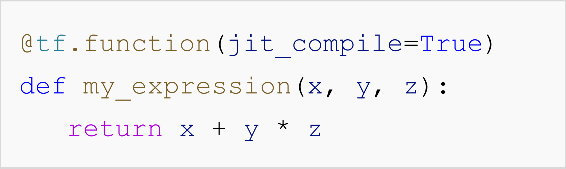
Not just Machine Learning!

Can be used for any fast numerical computations

Array-oriented backends

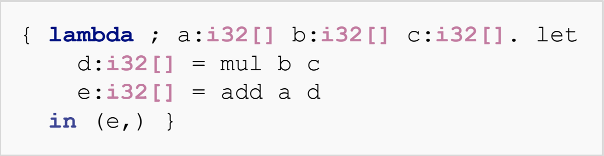
Tools from the ML and data science community that allow us to **outsource heavy computations**:

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

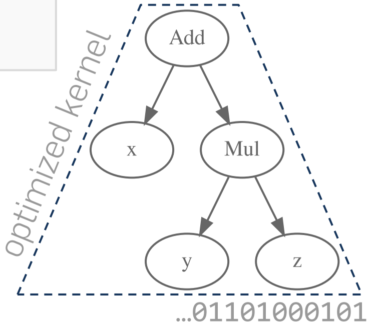


```
@tf.function(jit_compile=True)
def my_expression(x, y, z):
    return x + y * z
```

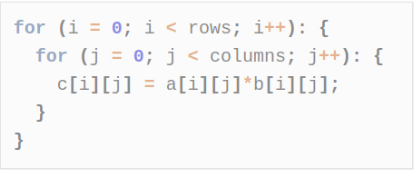
Converted to device-agnostic XLA code



```
{ 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];
  }
}
```

Array-oriented backends

Tools from the ML and data science community that allow us to **outsource heavy computations**:

- 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
```

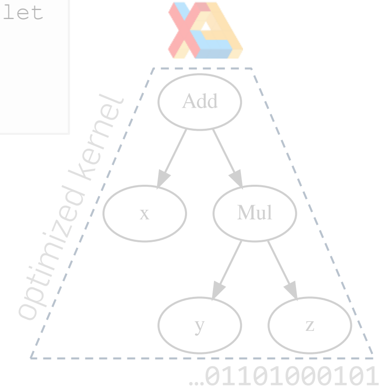
Usually all that the user needs to do

Converted to device-agnostic XLA code

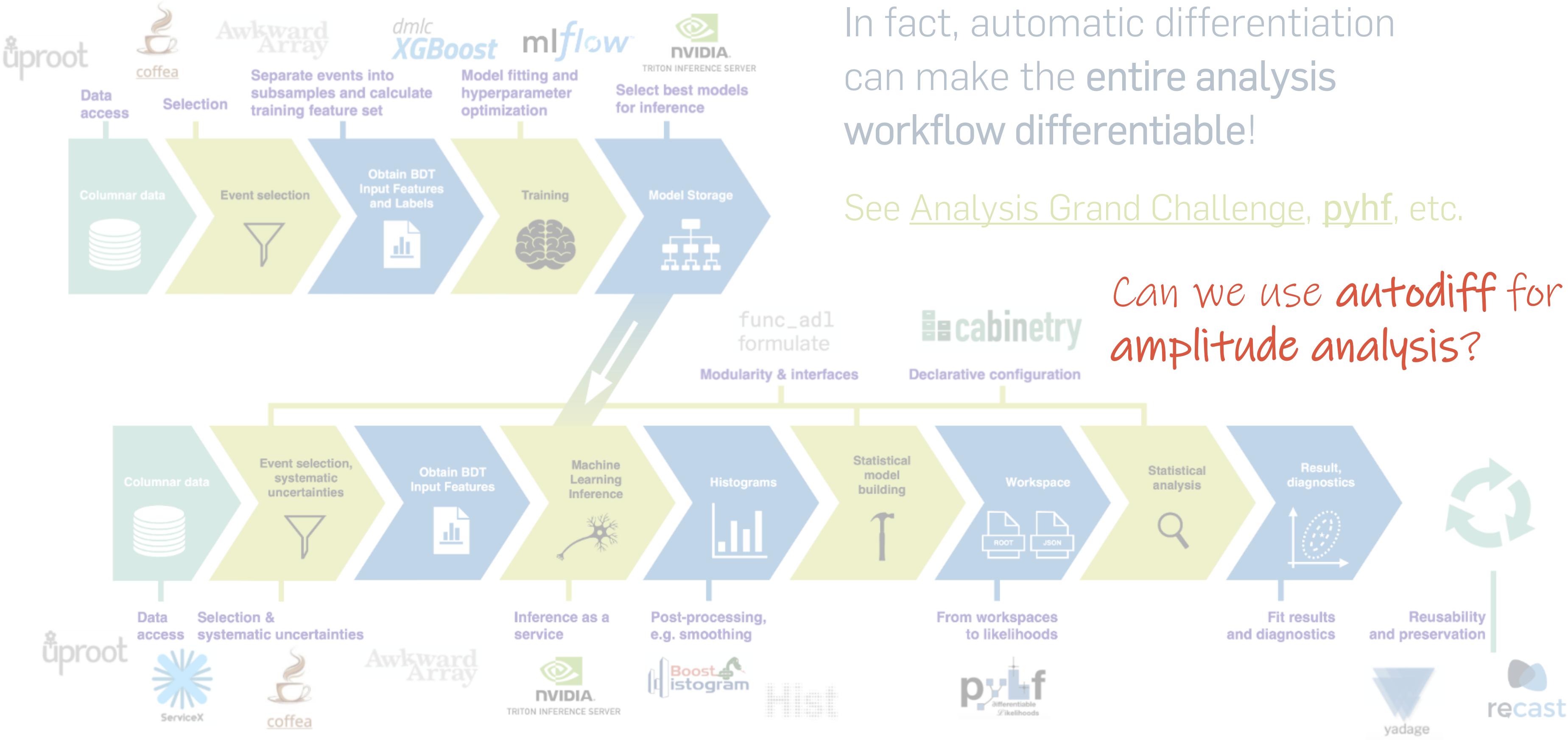
```
{ lambda ; a:i32[] b:i32[] c:i32[]. let
    d:i32[] = mul b c
    e:i32[] = add a d
  in (e,) }
```

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];
  }
}
```



Array-oriented backends | Automatic differentiation



uproot
coffea
Awkward Array
dmlc XGBoost
mlflow
NVIDIA TRITON INFERENCE SERVER

Data access
Selection
Separate events into subsamples and calculate training feature set
Model fitting and hyperparameter optimization
Select best models for inference

Columnar data
Event selection
Obtain BDT Input Features and Labels
Training
Model Storage

func_adl
formulate
Modularity & interfaces
cabinetry
Declarative configuration

Columnar data
Event selection, systematic uncertainties
Obtain BDT Input Features
Machine Learning Inference
Histograms
Statistical model building
Workspace
ROOT
JSON
Statistical analysis
Result, diagnostics

Data access
Selection & systematic uncertainties
Inference as a service
Post-processing, e.g. smoothing
From workspaces to likelihoods
Fit results and diagnostics
Reusability and preservation

uproot
ServiceX
coffea
Awkward Array
NVIDIA TRITON INFERENCE SERVER
Boost histogram
Hist
pyhf
differentiable likelihoods
yadage
recast

In fact, automatic differentiation can make the **entire analysis workflow differentiable**!

See Analysis Grand Challenge, pyhf, etc.

Can we use **autodiff** for amplitude analysis?

Mission: bring code closer to theory

✓ Outsource computations to **array-oriented backends** from the ML and data science community

$ax^2 + by^2$

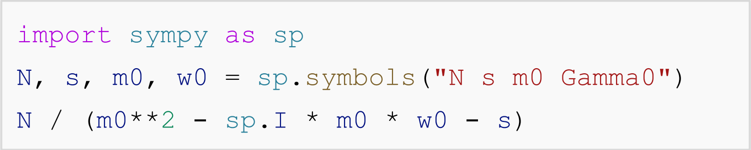
Flexibility through a **Computer Algebra System**

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
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



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

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

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

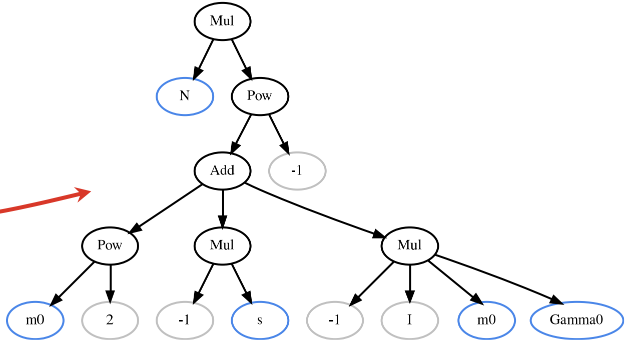
- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)

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

SymPy

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

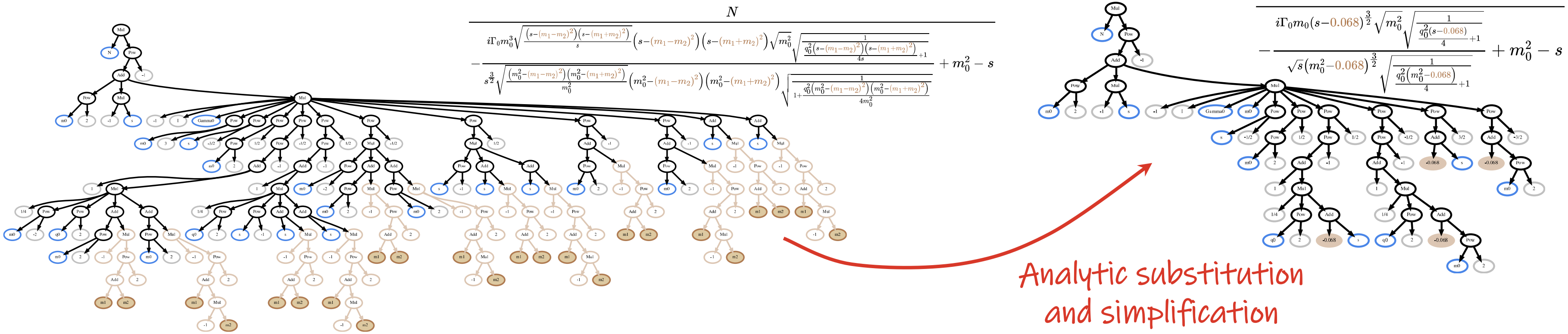
CAS represents expression as a tree



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
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



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

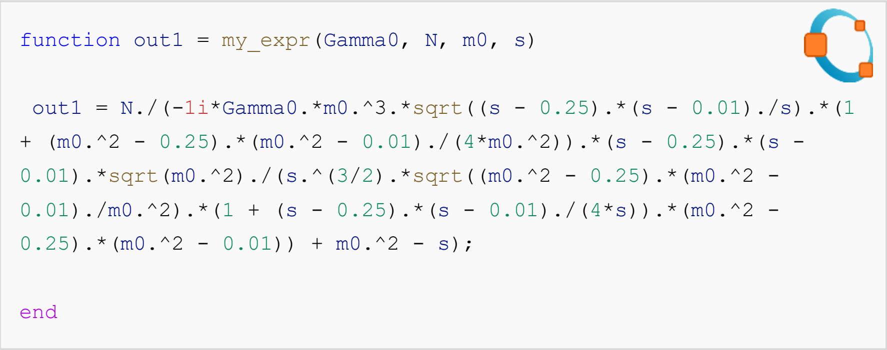
$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$

Analytic substitution
and simplification

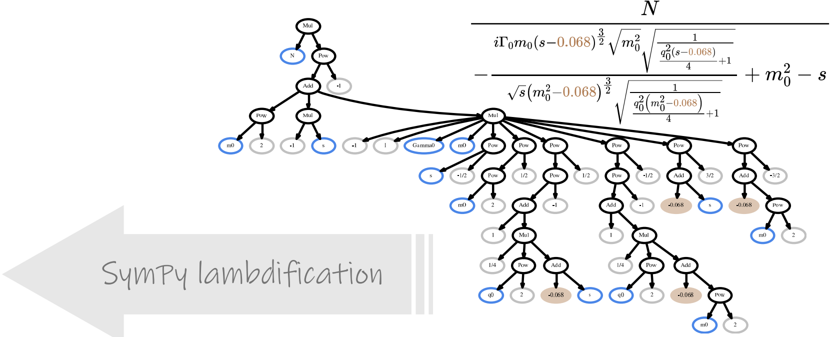
Symbolic amplitude models

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
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



```
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
```

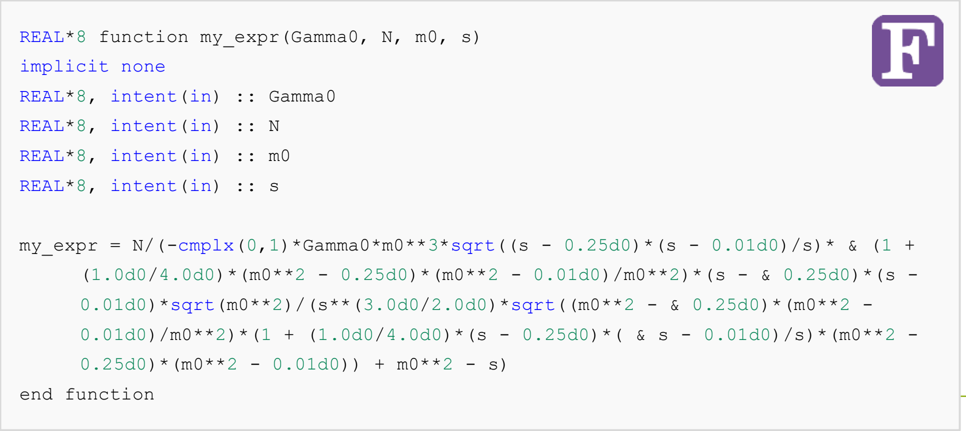


$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$

Symbolic amplitude models

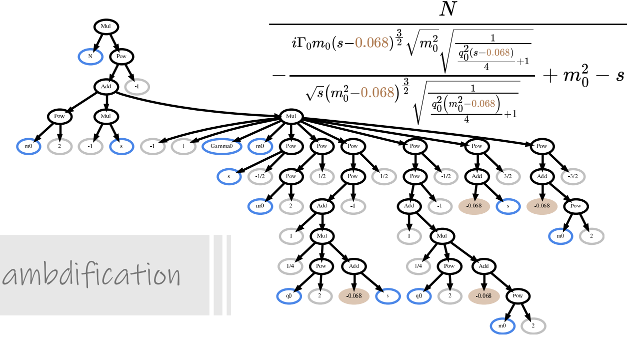
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



```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

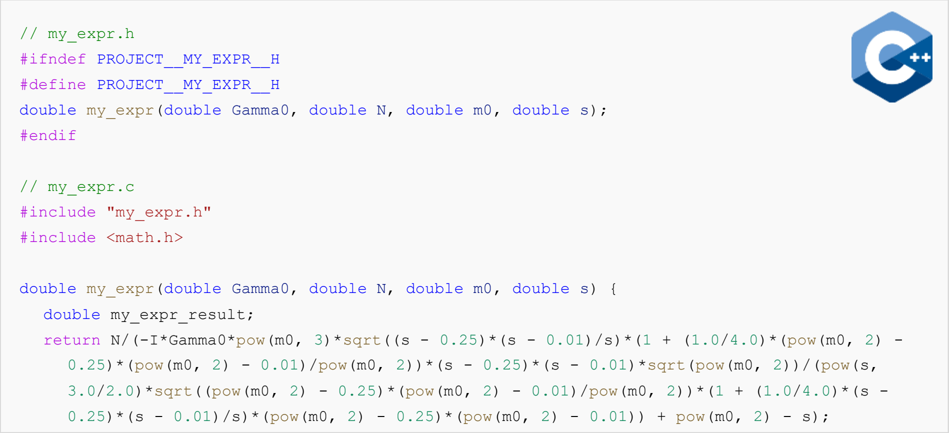


$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$

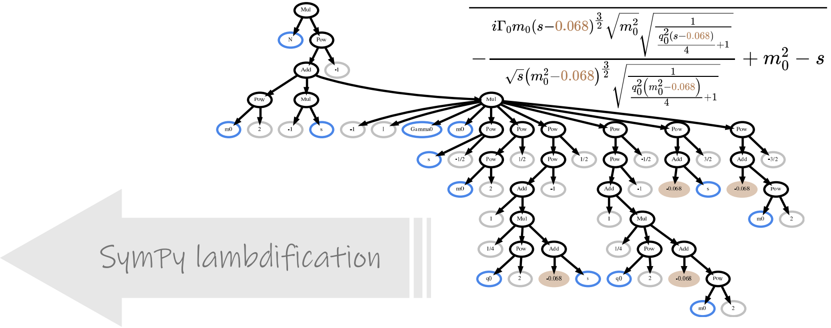
Symbolic amplitude models

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
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



```
// 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);
```

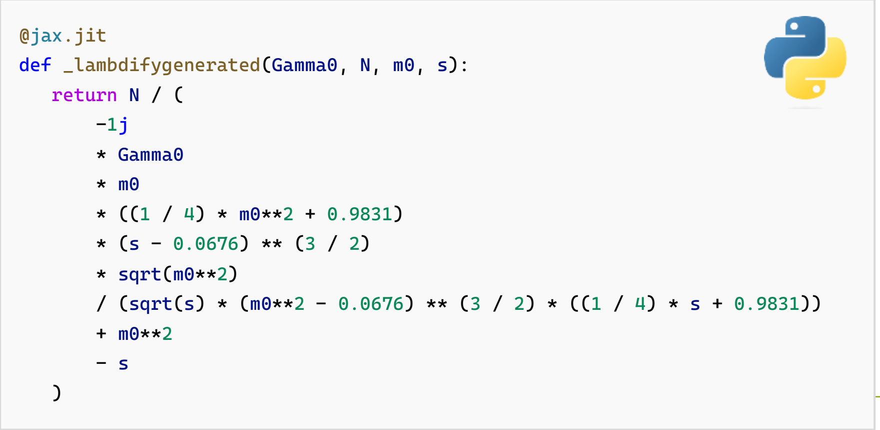


$$\frac{N}{-\frac{i\Gamma_0 m_0 (s-0.068)^{\frac{3}{2}}\sqrt{m_0^2}\sqrt{\frac{1}{\frac{q_0^2(s-0.068)}{4}+1}}}{\sqrt{s}\left(m_0^2-0.068\right)^{\frac{3}{2}}\sqrt{\frac{1}{\frac{q_0^2\left(m_0^2-0.068\right)}{4}+1}}}+m_0^2-s}$$

Symbolic amplitude models

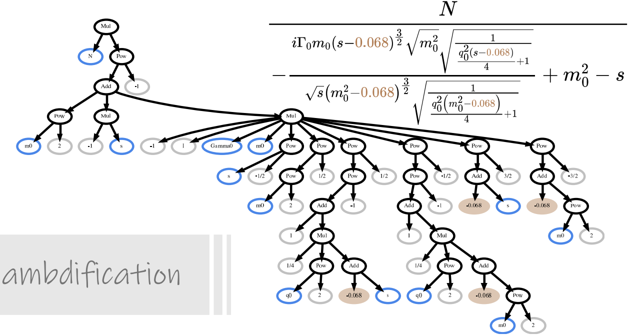
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



```
@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



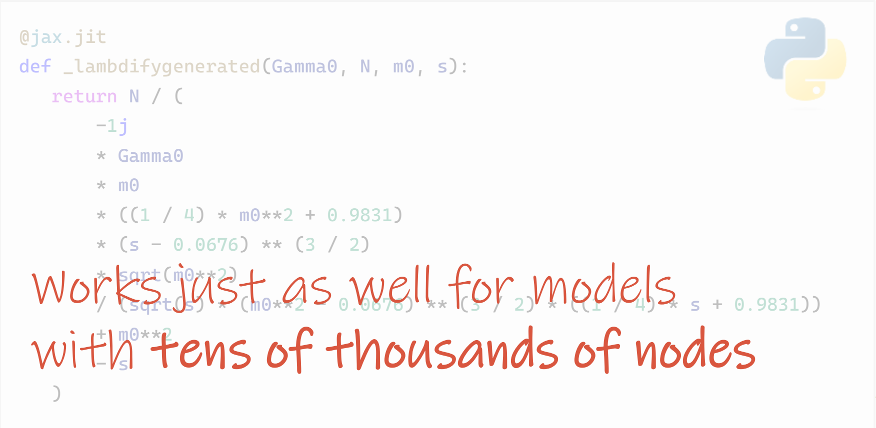
$$\frac{N}{- \frac{i \Gamma_{0} m_{0} \left(s - 0.068\right)^{\frac{3}{2}} \sqrt{m_{0}^{2}} \sqrt{\frac{1}{\frac{q_{0}^{2} \left(s - 0.068\right)}{4} + 1}}}{\sqrt{s} \left(m_{0}^{2} - 0.068\right)^{\frac{3}{2}} \sqrt{\frac{1}{\frac{q_{0}^{2} \left(m_{0}^{2} - 0.068\right)}{4} + 1}}} + m_{0}^{2} - s}$$

Same principle for **serialization!**

Symbolic amplitude models

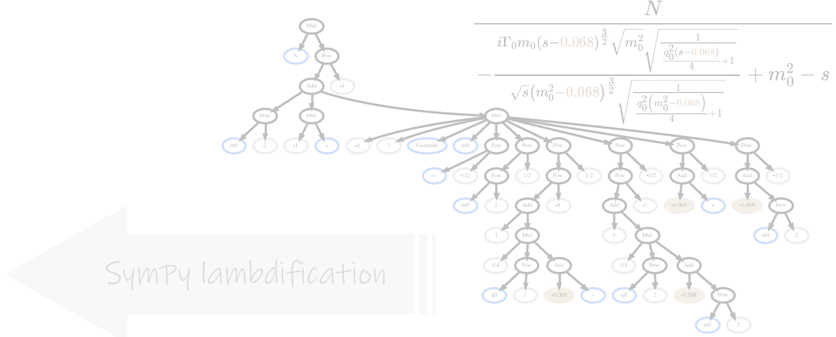
A new technique: formulate your amplitude model with a Computer Algebra System

- Transparency: inspect the math as you formulate the model
- Flexibility: modify the model with analytic substitutions
- Performance: simplify expressions algebraically
- Code generation: symbolic model as template to computational back-ends (SSoT)



```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
    )
```

Works just as well for models with **tens of thousands of nodes**



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

Same principle for **serialization!**

Mission: bring code closer to theory

- ✓ Outsource computations to **array-oriented backends** from the ML and data science community
- ✓ $ax^2 + by^2$ Flexibility through a **Computer Algebra System**
- Academic continuity through **living documentation**

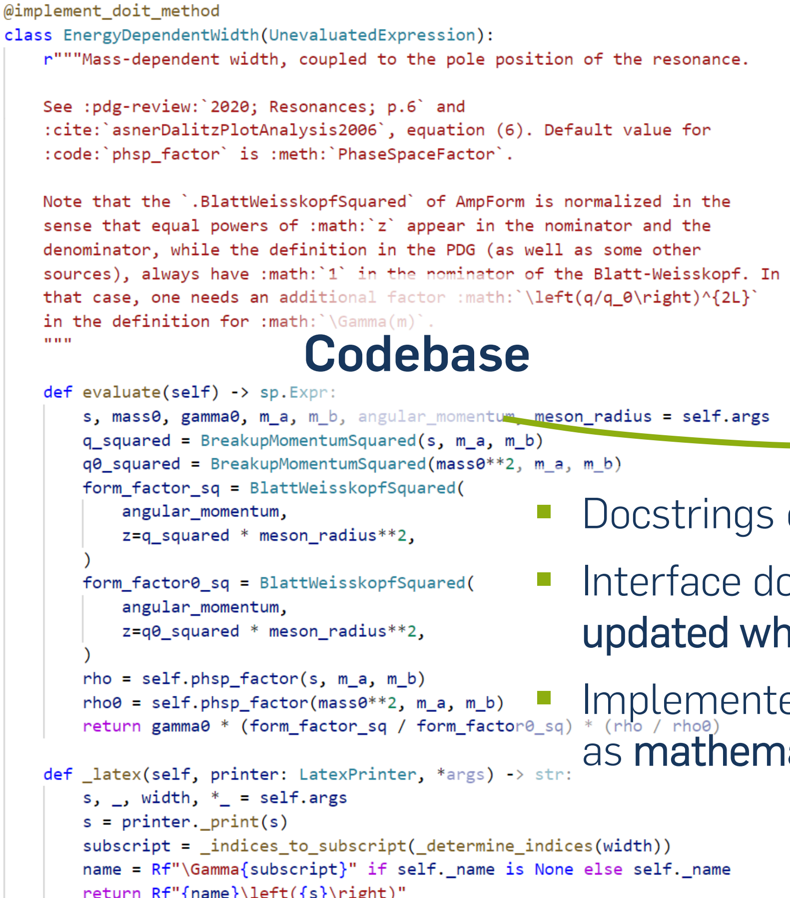
Living documentation

Python makes it easy to document codebases and analysis workflows:

- Jupyter notebooks
  - Interactive workflow
  - Surround code and results with explanatory text
- Documentation generation with Sphinx:
  - Automatic interface documentation with Sphinx
  - Render notebooks and codebase as HTML pages
  - Large ecosystem of powerful extensions

→ **Symbolic expressions turn this into a self-documenting workflow**

Living documentation

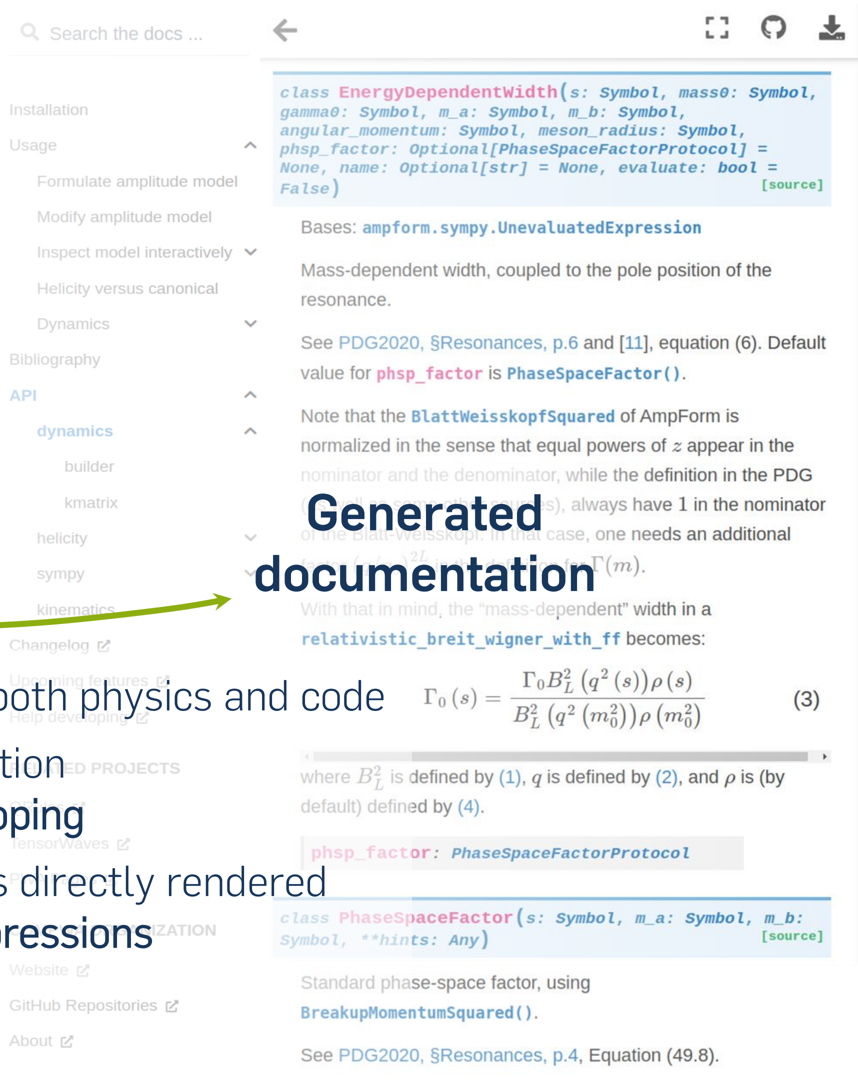


```
@implement_doit_method
class EnergyDependentWidth(UnevaluatedExpression):
    r"""Mass-dependent width, coupled to the pole position of the resonance.

    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)"
```

Codebase

Generated documentation

- Docstrings explains both physics and code
- Interface documentation **updated while developing**
- Implemented physics directly rendered as **mathematical expressions**



class `EnergyDependentWidth`(*s: Symbol, mass0: Symbol, gamma0: Symbol, m_a: Symbol, m_b: Symbol, angular_momentum: Symbol, meson_radius: Symbol, phsp_factor: Optional[PhaseSpaceFactorProtocol] = None, name: Optional[str] = None, evaluate: bool = False*) [source]

Bases: `ampform.sympy.UnevaluatedExpression`

Mass-dependent width, coupled to the pole position of the resonance.

See PDG2020, §Resonances, p.6 and [11], equation (6). Default value for `phsp_factor` is `PhaseSpaceFactor()`.

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

With that in mind, the "mass-dependent" width in a `relativistic_breit_wigner_with_ff` becomes:

$$\Gamma_0(s) = \frac{\Gamma_0 B_L^2\left(q^2(s)\right)\rho(s)}{B_L^2\left(q^2\left(m_0^2\right)\right)\rho\left(m_0^2\right)} \tag{3}$$

where B_L^2 is defined by (1), q is defined by (2), and ρ is (by default) defined by (4).

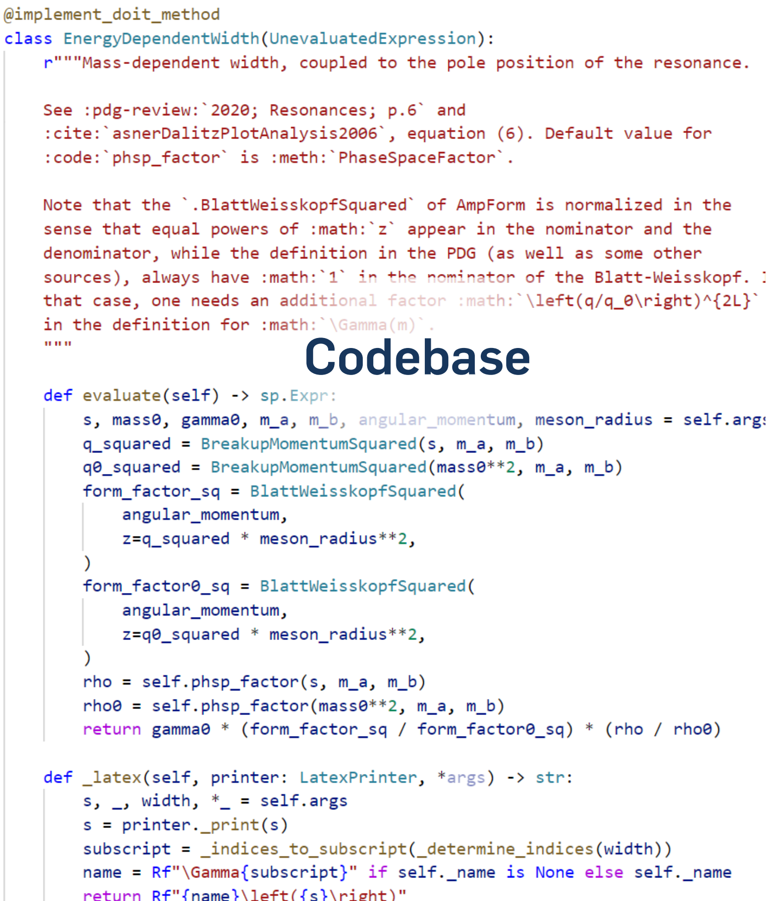
`phsp_factor`: *PhaseSpaceFactorProtocol*

class `PhaseSpaceFactor`(*s: Symbol, m_a: Symbol, m_b: Symbol, **hints: Any*) [source]

Standard phase-space factor, using `BreakupMomentumSquared()`.

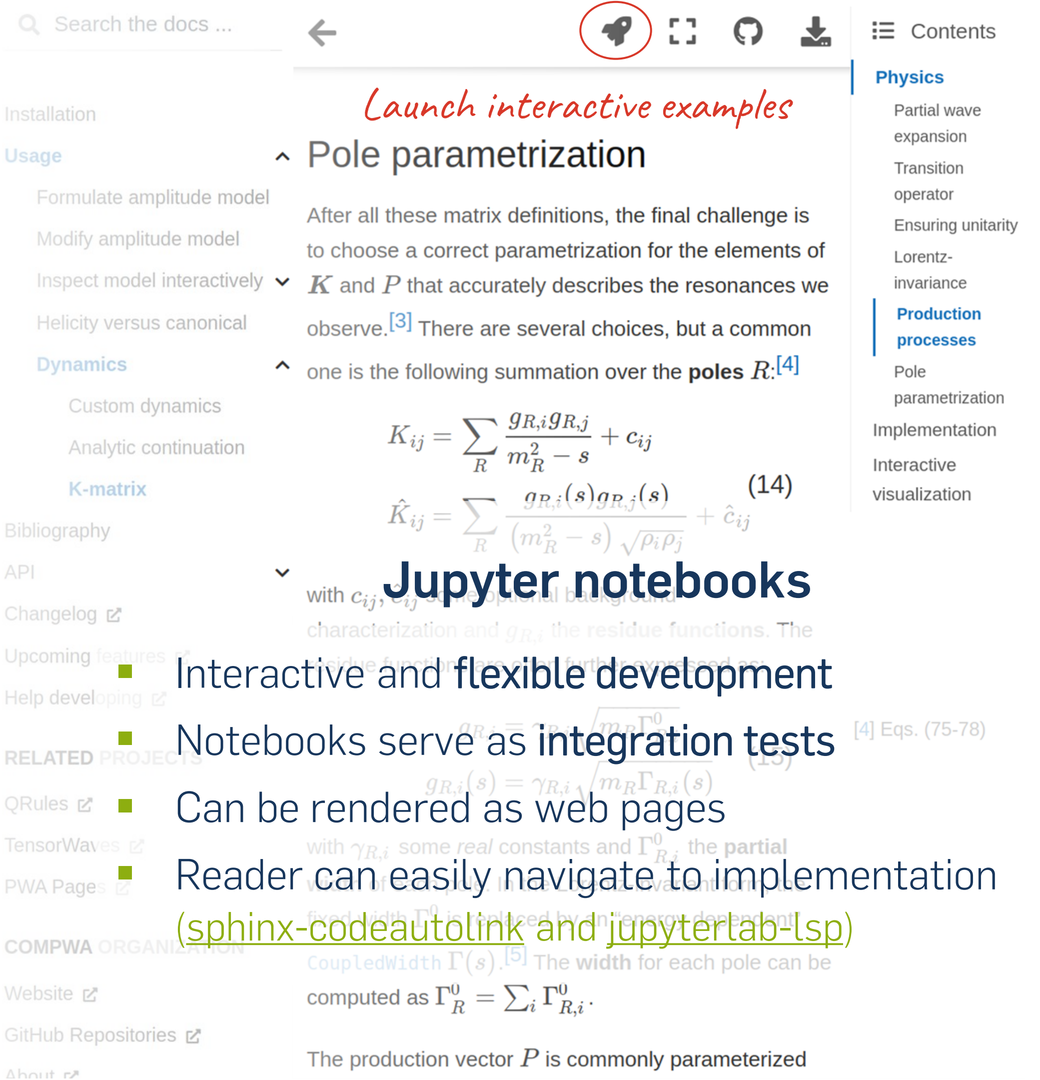
See PDG2020, §Resonances, p.4, Equation (49.8).

Living documentation



Codebase

```
@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.
    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)"
```



Launch interactive examples

Jupyter notebooks

Pole parametrization

After all these matrix definitions, the final challenge is to choose a correct parametrization for the elements of K and 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]

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

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

$$
\begin{aligned}
g_{R,i} &= \gamma_{R,i}\sqrt{m_R\Gamma^0_{R,i}} \\
g_{R,i}(s) &= \gamma_{R,i}\sqrt{m_R\Gamma_{R,i}(s)}
\end{aligned}
\qquad (15)
$$

[4] Eqs. (75-78)

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

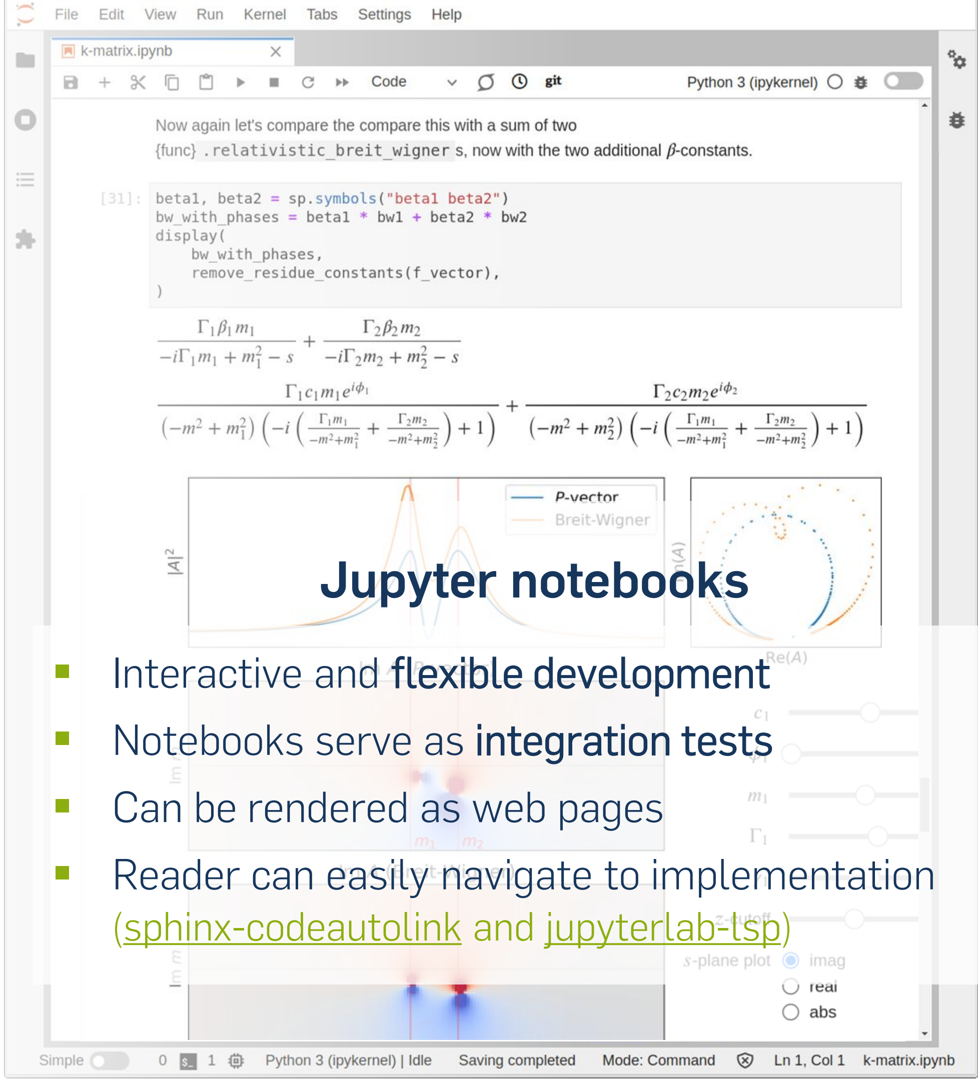
- Interactive and **flexible development**
- Notebooks serve as **integration tests**
- Can be rendered as web pages
- Reader can easily navigate to implementation (sphinx-codeautolink and jupyterlab-lsp)

Living documentation

Codebase

```
@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



Now again let's compare the compare this with a sum of two {func} `.relativistic_breit_wigner` s, now with the two additional β-constants.

```
[31]: beta1, beta2 = sp.symbols("beta1 beta2")
      bw_with_phases = beta1 * bw1 + beta2 * bw2
      display(
          bw_with_phases,
          remove_residue_constants(f_vector),
      )
```

$$\frac{\Gamma_1\beta_1 m_1}{-i\Gamma_1 m_1 + m_1^2 - s} + \frac{\Gamma_2\beta_2 m_2}{-i\Gamma_2 m_2 + m_2^2 - s}$$

$$\frac{\Gamma_1 c_1 m_1 e^{i\phi_1}}{\left(-m^2+m_1^2\right)\left(-i\left(\frac{\Gamma_1 m_1}{-m^2+m_1^2}+\frac{\Gamma_2 m_2}{-m^2+m_2^2}\right)+1\right)} + \frac{\Gamma_2 c_2 m_2 e^{i\phi_2}}{\left(-m^2+m_2^2\right)\left(-i\left(\frac{\Gamma_1 m_1}{-m^2+m_1^2}+\frac{\Gamma_2 m_2}{-m^2+m_2^2}\right)+1\right)}$$

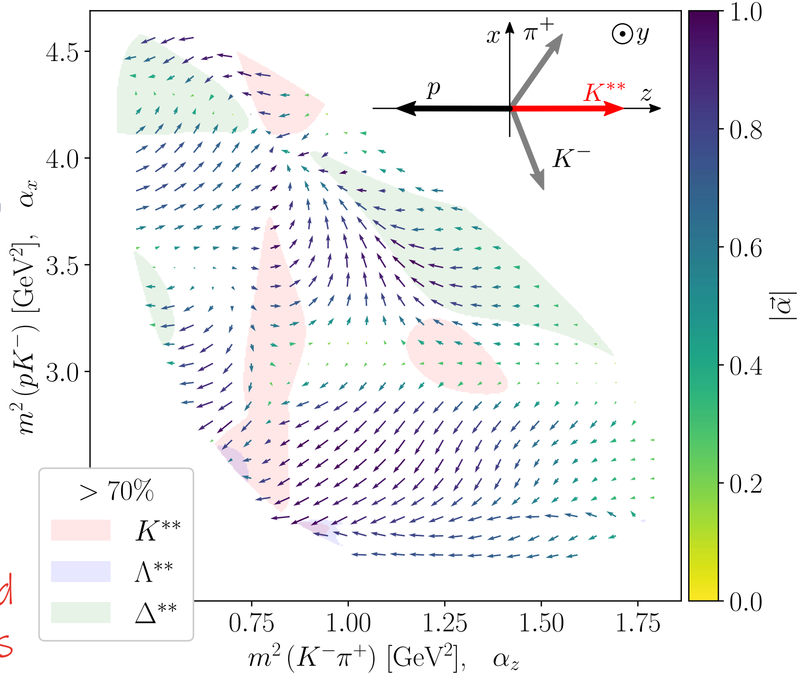
- Interactive and **flexible development**
- Notebooks serve as **integration tests**
- Can be rendered as web pages
- Reader can easily navigate to implementation (sphinx-codeautolink and jupyterlab-lsp)

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action

Symbolic amplitude models powered a recent study by LHCb [10.1007/JHEP07(2023)228]

- Dalitz-plot decomposition separates angular d.o.f. from dynamics d.o.f.
- Resulting amplitudes can be used to compute a polarimeter vector field for propagating polarization info
- Symbolic amplitude models flexible enough for transformation into polarimeter vector field
- Perfect example of self-documenting workflow



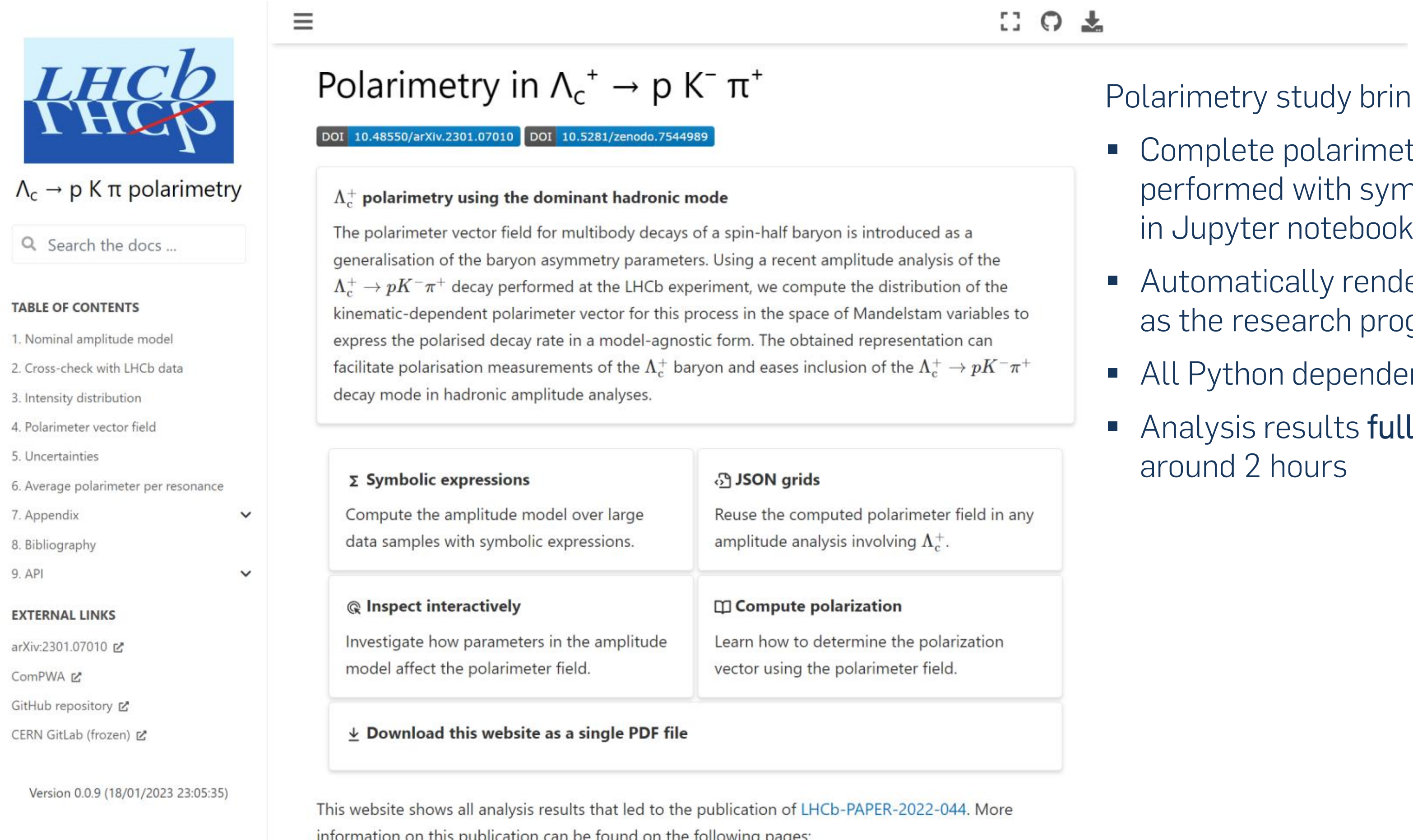
$$|\mathcal{M}(\phi,\theta,\chi,\kappa)|^2 = I_0(\kappa)\left(1+\sum_{i,j=1}^{3} P_i R_{ij}(\phi,\theta,\chi)\alpha_j(\kappa)\right)$$

$$\vec{\alpha}(\kappa) = \sum_{\nu',\nu,\{\lambda\}} A^*_{\nu',\{\lambda\}}\vec{\sigma}_{\nu',\nu}A_{\nu,\{\lambda\}} \,/\, I_0(\kappa)$$

powered by symbolics

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action



$\Lambda_c \to p\,K\,\pi$ polarimetry

Search the docs ...

TABLE OF CONTENTS

EXTERNAL LINKS

- arXiv:2301.07010
- ComPWA
- GitHub repository
- CERN GitLab (frozen)

Version 0.0.9 (18/01/2023 23:05:35)

Polarimetry in $\Lambda_c^+ \to p\,K^-\,\pi^+$

DOI 10.48550/arXiv.2301.07010 DOI 10.5281/zenodo.7544989

Λ_c^+ polarimetry using the dominant hadronic mode

The polarimeter vector field for multibody decays of a spin-half baryon is introduced as a generalisation of the baryon asymmetry parameters. Using a recent amplitude analysis of the $\Lambda_c^+ \to pK^-\pi^+$ decay performed at the LHCb experiment, we compute the distribution of the kinematic-dependent polarimeter vector for this process in the space of Mandelstam variables to express the polarised decay rate in a model-agnostic form. The obtained representation can facilitate polarisation measurements of the Λ_c^+ baryon and eases inclusion of the $\Lambda_c^+ \to pK^-\pi^+$ decay mode in hadronic amplitude analyses.

Σ Symbolic expressions

Compute the amplitude model over large data samples with symbolic expressions.

JSON grids

Reuse the computed polarimeter field in any amplitude analysis involving Λ_c^+.

Inspect interactively

Investigate how parameters in the amplitude model affect the polarimeter field.

Compute polarization

Learn how to determine the polarization vector using the polarimeter field.

Download this website as a single PDF file

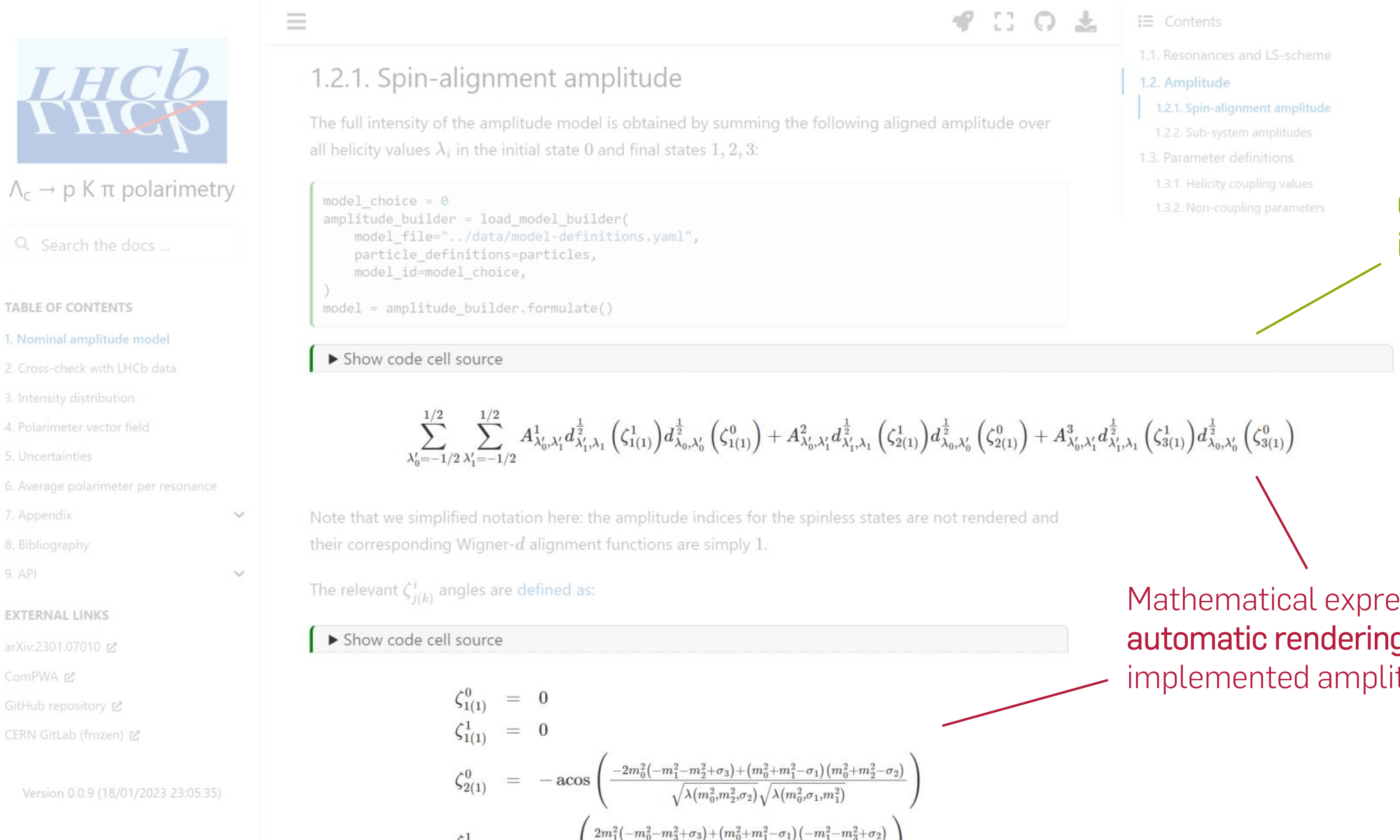
This website shows all analysis results that led to the publication of LHCb-PAPER-2022-044. More information on this publication can be found on the following pages:

Polarimetry study brings it all together:

- Complete polarimetry analysis performed with symbolic expressions in Jupyter notebooks
- Automatically rendered as webpages as the research progressed
- All Python dependencies are pinned
- Analysis results **fully reproducible** in around 2 hours

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action



$\Lambda_c \to p\,K\,\pi$ polarimetry

Search the docs ...

TABLE OF CONTENTS

1. Nominal amplitude model
2. Cross-check with LHCb data
3. Intensity distribution
4. Polarimeter vector field
5. Uncertainties
6. Average polarimeter per resonance
7. Appendix
8. Bibliography
9. API

EXTERNAL LINKS

arXiv:2301.07010

ComPWA

GitHub repository

CERN GitLab (frozen)

Version 0.0.9 (18/01/2023 23:05:35)

Contents

1.1. Resonances and LS-scheme

1.2. Amplitude

1.2.1. Spin-alignment amplitude

1.2.2. Sub-system amplitudes

1.3. Parameter definitions

1.3.1. Helicity coupling values

1.3.2. Non-coupling parameters

1.2.1. Spin-alignment amplitude

The full intensity of the amplitude model is obtained by summing the following aligned amplitude over all helicity values λ_i in the initial state 0 and final states $1, 2, 3$:

```
model_choice = 0
amplitude_builder = load_model_builder(
    model_file="../data/model-definitions.yaml",
    particle_definitions=particles,
    model_id=model_choice,
)
model = amplitude_builder.formulate()
```

▶ Show code cell source

$$\sum_{\lambda_0'=-1/2}^{1/2}\sum_{\lambda_1'=-1/2}^{1/2} A^{1}_{\lambda_0',\lambda_1'} d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^{1}_{1(1)}\right) d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^{0}_{1(1)}\right) + A^{2}_{\lambda_0',\lambda_1'} d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^{1}_{2(1)}\right) d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^{0}_{2(1)}\right) + A^{3}_{\lambda_0',\lambda_1'} d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^{1}_{3(1)}\right) d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^{0}_{3(1)}\right)$$

Note that we simplified notation here: the amplitude indices for the spinless states are not rendered and their corresponding Wigner-d alignment functions are simply 1.

The relevant $\zeta^i_{j(k)}$ angles are defined as:

▶ Show code cell source

$$\begin{aligned}
\zeta^{0}_{1(1)} &= 0 \\
\zeta^{1}_{1(1)} &= 0 \\
\zeta^{0}_{2(1)} &= -\operatorname{acos}\left(\frac{-2m_0^2\left(-m_1^2-m_2^2+\sigma_3\right)+\left(m_0^2+m_1^2-\sigma_1\right)\left(m_0^2+m_2^2-\sigma_2\right)}{\sqrt{\lambda\left(m_0^2,m_2^2,\sigma_2\right)}\sqrt{\lambda\left(m_0^2,\sigma_1,m_1^2\right)}}\right)
\end{aligned}$$

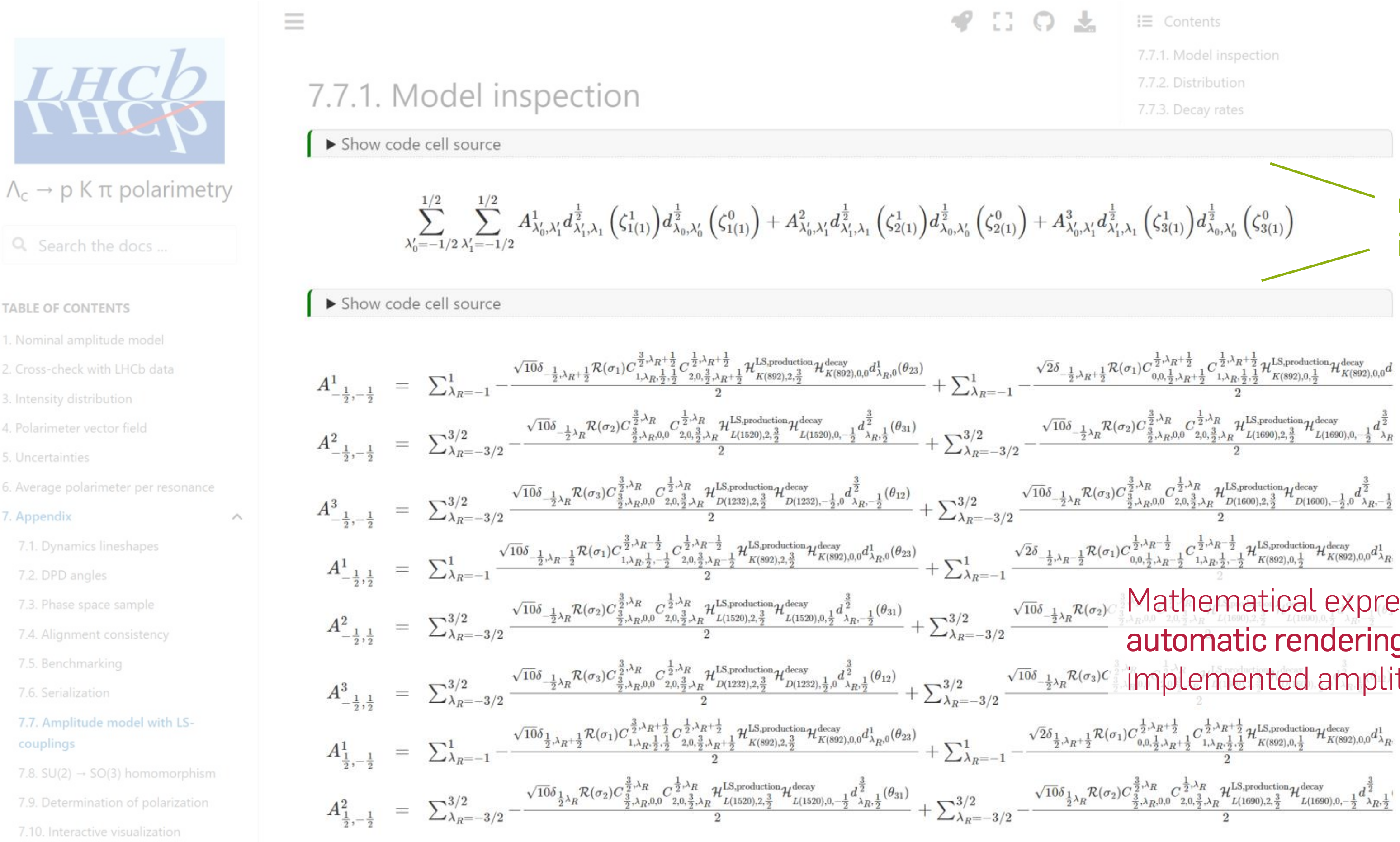
$$2m_1^2\left(-m_0^2-m_3^2+\sigma_3\right)+\left(m_0^2+m_1^2-\sigma_1\right)\left(-m_1^2-m_3^2+\sigma_2\right)$$

Code can be inspected directly

Mathematical expressions are **automatic rendering** of the implemented amplitude models

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action



7.7.1. Model inspection

▶ Show code cell source

$$\sum_{\lambda_0'=-1/2}^{1/2}\sum_{\lambda_1'=-1/2}^{1/2} A^1_{\lambda_0',\lambda_1'}d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^1_{1(1)}\right)d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^0_{1(1)}\right)+A^2_{\lambda_0',\lambda_1'}d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^1_{2(1)}\right)d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^0_{2(1)}\right)+A^3_{\lambda_0',\lambda_1'}d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^1_{3(1)}\right)d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^0_{3(1)}\right)$$

▶ Show code cell source

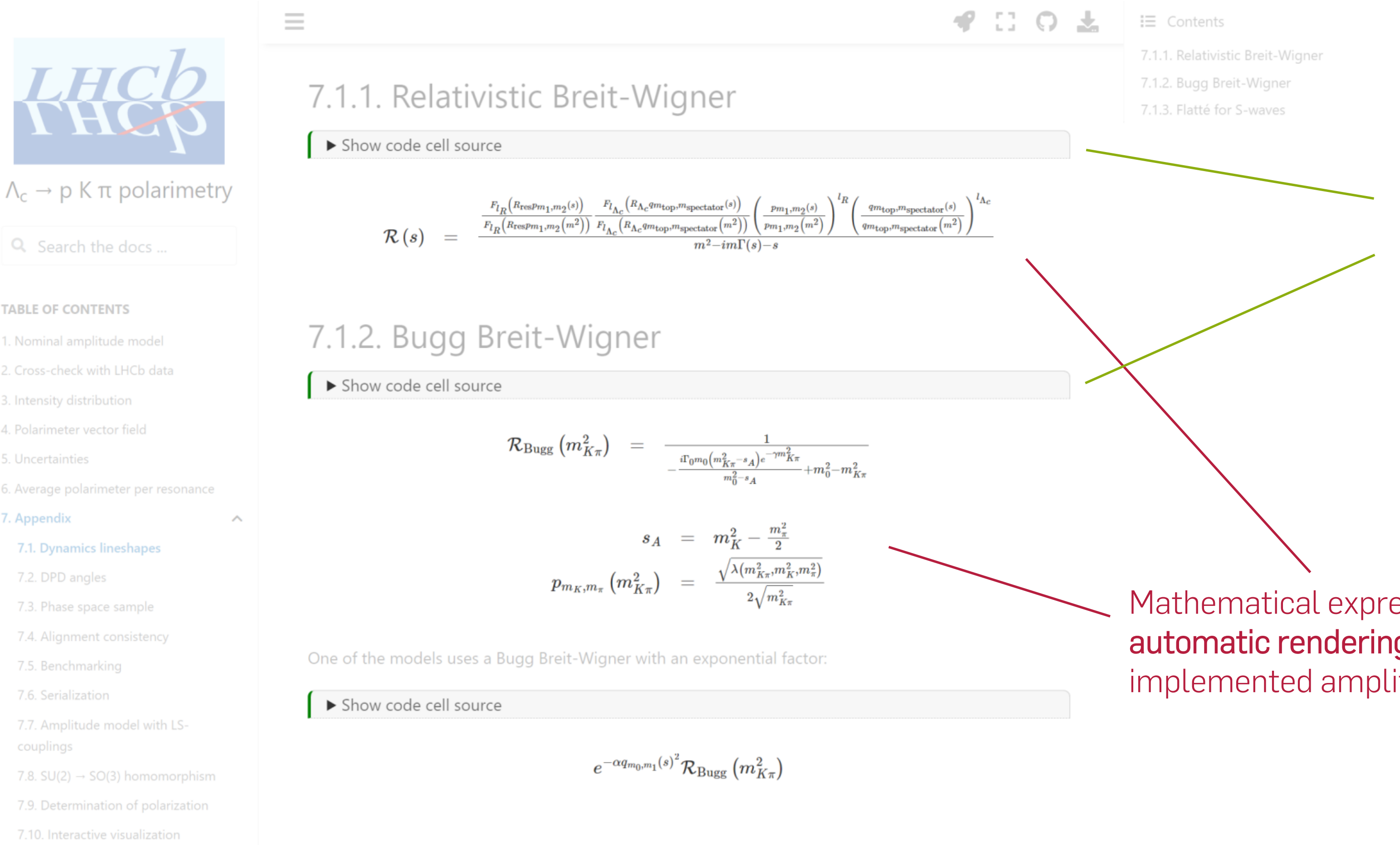
$$\begin{aligned}
A^1{}_{-\frac{1}{2},-\frac{1}{2}} &= \sum_{\lambda_R=-1}^{1} -\frac{\sqrt{10}\delta_{-\frac{1}{2},\lambda_R+\frac{1}{2}}\mathcal{R}(\sigma_1)C^{\frac{3}{2},\lambda_R+\frac{1}{2}}_{1,\lambda_R,\frac{1}{2},\frac{1}{2}}C^{\frac{1}{2},\lambda_R+\frac{1}{2}}_{2,0,\frac{3}{2},\lambda_R+\frac{1}{2}}\mathcal{H}^{\mathrm{LS,production}}_{K(892),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{K(892),0,0}d^{1}_{\lambda_R,0}(\theta_{23})}{2}+\sum_{\lambda_R=-1}^{1}\\
A^2{}_{-\frac{1}{2},-\frac{1}{2}} &= \sum_{\lambda_R=-3/2}^{3/2} -\frac{\sqrt{10}\delta_{-\frac{1}{2}\lambda_R}\mathcal{R}(\sigma_2)C^{\frac{3}{2},\lambda_R}_{\frac{3}{2},\lambda_R,0,0}C^{\frac{1}{2},\lambda_R}_{2,0,\frac{3}{2},\lambda_R}\mathcal{H}^{\mathrm{LS,production}}_{L(1520),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{L(1520),0,-\frac{1}{2}}d^{\frac{3}{2}}_{\lambda_R,\frac{1}{2}}(\theta_{31})}{2}+\sum_{\lambda_R=-3/2}^{3/2}\\
A^3{}_{-\frac{1}{2},-\frac{1}{2}} &= \sum_{\lambda_R=-3/2}^{3/2} \frac{\sqrt{10}\delta_{-\frac{1}{2}\lambda_R}\mathcal{R}(\sigma_3)C^{\frac{3}{2},\lambda_R}_{\frac{3}{2},\lambda_R,0,0}C^{\frac{1}{2},\lambda_R}_{2,0,\frac{3}{2},\lambda_R}\mathcal{H}^{\mathrm{LS,production}}_{D(1232),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{D(1232),-\frac{1}{2},0}d^{\frac{3}{2}}_{\lambda_R,-\frac{1}{2}}(\theta_{12})}{2}+\sum_{\lambda_R=-3/2}^{3/2}\\
A^1{}_{-\frac{1}{2},\frac{1}{2}} &= \sum_{\lambda_R=-1}^{1} \frac{\sqrt{10}\delta_{-\frac{1}{2},\lambda_R-\frac{1}{2}}\mathcal{R}(\sigma_1)C^{\frac{3}{2},\lambda_R-\frac{1}{2}}_{1,\lambda_R,\frac{1}{2},-\frac{1}{2}}C^{\frac{1}{2},\lambda_R-\frac{1}{2}}_{2,0,\frac{3}{2},\lambda_R-\frac{1}{2}}\mathcal{H}^{\mathrm{LS,production}}_{K(892),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{K(892),0,0}d^{1}_{\lambda_R,0}(\theta_{23})}{2}+\sum_{\lambda_R=-1}^{1}\\
A^2{}_{-\frac{1}{2},\frac{1}{2}} &= \sum_{\lambda_R=-3/2}^{3/2} \frac{\sqrt{10}\delta_{-\frac{1}{2}\lambda_R}\mathcal{R}(\sigma_2)C^{\frac{3}{2},\lambda_R}_{\frac{3}{2},\lambda_R,0,0}C^{\frac{1}{2},\lambda_R}_{2,0,\frac{3}{2},\lambda_R}\mathcal{H}^{\mathrm{LS,production}}_{L(1520),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{L(1520),0,\frac{1}{2}}d^{\frac{3}{2}}_{\lambda_R,-\frac{1}{2}}(\theta_{31})}{2}+\sum_{\lambda_R=-3/2}^{3/2}\\
A^3{}_{-\frac{1}{2},\frac{1}{2}} &= \sum_{\lambda_R=-3/2}^{3/2} \frac{\sqrt{10}\delta_{-\frac{1}{2}\lambda_R}\mathcal{R}(\sigma_3)C^{\frac{3}{2},\lambda_R}_{\frac{3}{2},\lambda_R,0,0}C^{\frac{1}{2},\lambda_R}_{2,0,\frac{3}{2},\lambda_R}\mathcal{H}^{\mathrm{LS,production}}_{D(1232),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{D(1232),\frac{1}{2},0}d^{\frac{3}{2}}_{\lambda_R,\frac{1}{2}}(\theta_{12})}{2}+\sum_{\lambda_R=-3/2}^{3/2}\\
A^1{}_{\frac{1}{2},-\frac{1}{2}} &= \sum_{\lambda_R=-1}^{1} -\frac{\sqrt{10}\delta_{\frac{1}{2},\lambda_R+\frac{1}{2}}\mathcal{R}(\sigma_1)C^{\frac{3}{2},\lambda_R+\frac{1}{2}}_{1,\lambda_R,\frac{1}{2},\frac{1}{2}}C^{\frac{1}{2},\lambda_R+\frac{1}{2}}_{2,0,\frac{3}{2},\lambda_R+\frac{1}{2}}\mathcal{H}^{\mathrm{LS,production}}_{K(892),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{K(892),0,0}d^{1}_{\lambda_R,0}(\theta_{23})}{2}+\sum_{\lambda_R=-1}^{1}\\
A^2{}_{\frac{1}{2},-\frac{1}{2}} &= \sum_{\lambda_R=-3/2}^{3/2} -\frac{\sqrt{10}\delta_{\frac{1}{2}\lambda_R}\mathcal{R}(\sigma_2)C^{\frac{3}{2},\lambda_R}_{\frac{3}{2},\lambda_R,0,0}C^{\frac{1}{2},\lambda_R}_{2,0,\frac{3}{2},\lambda_R}\mathcal{H}^{\mathrm{LS,production}}_{L(1520),2,\frac{3}{2}}\mathcal{H}^{\mathrm{decay}}_{L(1520),0,-\frac{1}{2}}d^{\frac{3}{2}}_{\lambda_R,\frac{1}{2}}(\theta_{31})}{2}+\sum_{\lambda_R=-3/2}^{3/2}
\end{aligned}$$

Code can be inspected directly

Mathematical expressions are **automatic rendering** of the implemented amplitude models

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action



$\Lambda_c \to p K \pi$ polarimetry

Search the docs ...

TABLE OF CONTENTS

Contents

7.1.1. Relativistic Breit-Wigner

▶ Show code cell source

$$\mathcal{R}\left(s\right) = \frac{\frac{F_{l_R}\left(R_\mathrm{res}p_{m_1,m_2}(s)\right)}{F_{l_R}\left(R_\mathrm{res}p_{m_1,m_2}\left(m^2\right)\right)}\frac{F_{l_{\Lambda_c}}\left(R_{\Lambda_c}q_{m_\mathrm{top},m_\mathrm{spectator}}(s)\right)}{F_{l_{\Lambda_c}}\left(R_{\Lambda_c}q_{m_\mathrm{top},m_\mathrm{spectator}}\left(m^2\right)\right)}\left(\frac{p_{m_1,m_2}(s)}{p_{m_1,m_2}\left(m^2\right)}\right)^{l_R}\left(\frac{q_{m_\mathrm{top},m_\mathrm{spectator}}(s)}{q_{m_\mathrm{top},m_\mathrm{spectator}}\left(m^2\right)}\right)^{l_{\Lambda_c}}}{m^2-im\Gamma(s)-s}$$

7.1.2. Bugg Breit-Wigner

▶ Show code cell source

$$\mathcal{R}_\mathrm{Bugg}\left(m_{K\pi}^2\right) = \frac{1}{-\frac{i\Gamma_0 m_0\left(m_{K\pi}^2-s_A\right)e^{-\gamma m_{K\pi}^2}}{m_0^2-s_A}+m_0^2-m_{K\pi}^2}$$

$$s_A = m_K^2 - \frac{m_\pi^2}{2}$$

$$p_{m_K,m_\pi}\left(m_{K\pi}^2\right) = \frac{\sqrt{\lambda(m_{K\pi}^2,m_K^2,m_\pi^2)}}{2\sqrt{m_{K\pi}^2}}$$

One of the models uses a Bugg Breit-Wigner with an exponential factor:

▶ Show code cell source

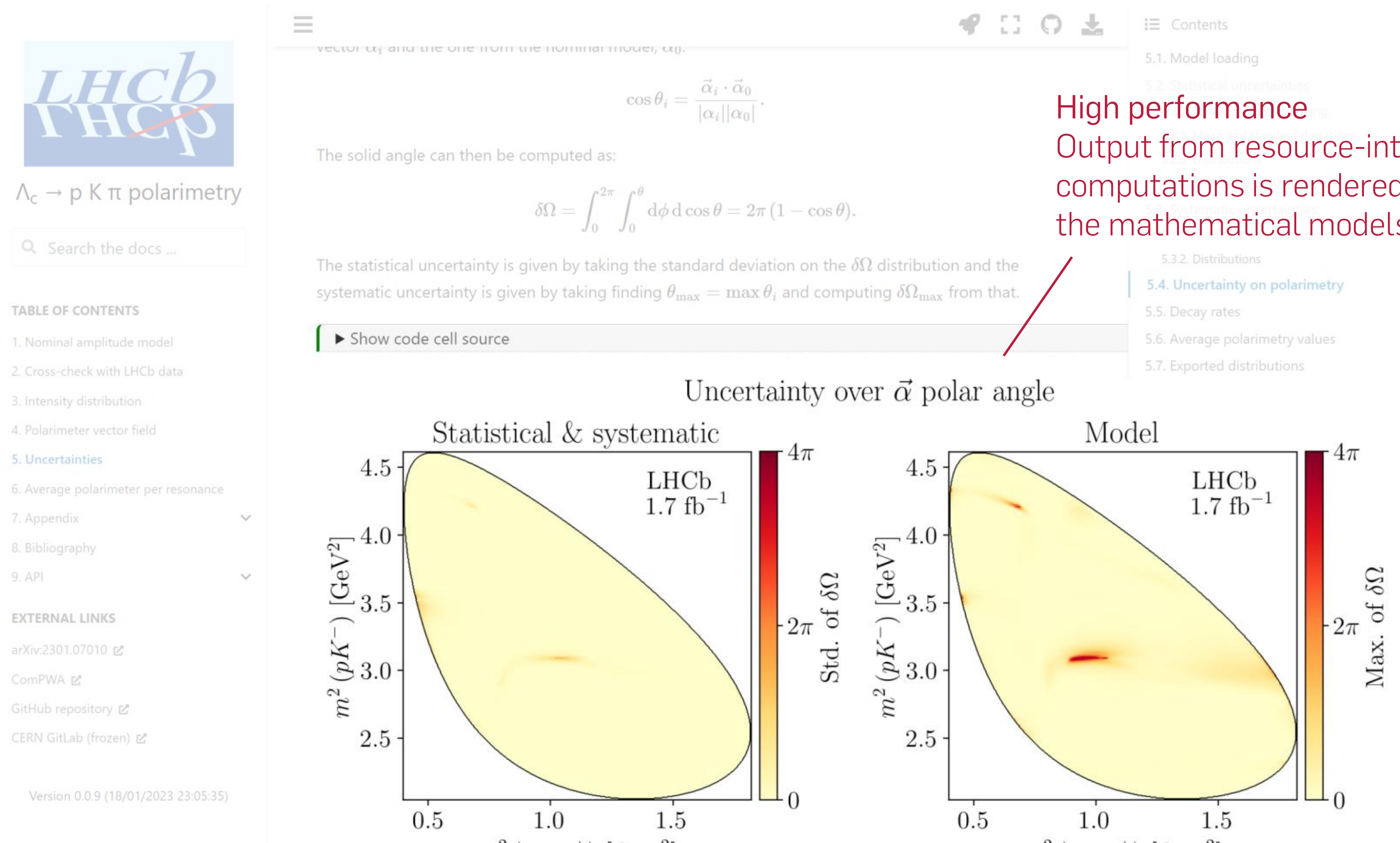
$$e^{-\alpha q_{m_0,m_1}(s)^2}\mathcal{R}_\mathrm{Bugg}\left(m_{K\pi}^2\right)$$

Code can be inspected directly

Mathematical expressions are **automatic rendering** of the implemented amplitude models

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action



$\Lambda_c \to p\,K\,\pi$ polarimetry

Search the docs ...

TABLE OF CONTENTS

1. Nominal amplitude model
2. Cross-check with LHCb data
3. Intensity distribution
4. Polarimeter vector field
5. Uncertainties
6. Average polarimeter per resonance
7. Appendix
8. Bibliography
9. API

EXTERNAL LINKS

- arXiv:2301.07010
- ComPWA
- GitHub repository
- CERN GitLab (frozen)

Version 0.0.9 (18/01/2023 23:05:35)

vector $\vec{\alpha}_i$ and the one from the nominal model, $\vec{\alpha}_0$:

$$\cos\theta_i = \frac{\vec{\alpha}_i \cdot \vec{\alpha}_0}{|\alpha_i||\alpha_0|}.$$

The solid angle can then be computed as:

$$\delta\Omega = \int_0^{2\pi}\int_0^{\theta} \mathrm{d}\phi\,\mathrm{d}\cos\theta = 2\pi\left(1-\cos\theta\right).$$

The statistical uncertainty is given by taking the standard deviation on the $\delta\Omega$ distribution and the systematic uncertainty is given by taking finding $\theta_\mathrm{max} = \max\theta_i$ and computing $\delta\Omega_\mathrm{max}$ from that.

▶ Show code cell source

Contents

5.1. Model loading
5.2. Statistical uncertainties
5.3. Systematic uncertainties
5.3.2. Distributions
5.4. Uncertainty on polarimetry
5.5. Decay rates
5.6. Average polarimetry values
5.7. Exported distributions

Uncertainty over $\vec{\alpha}$ polar angle

Statistical & systematic

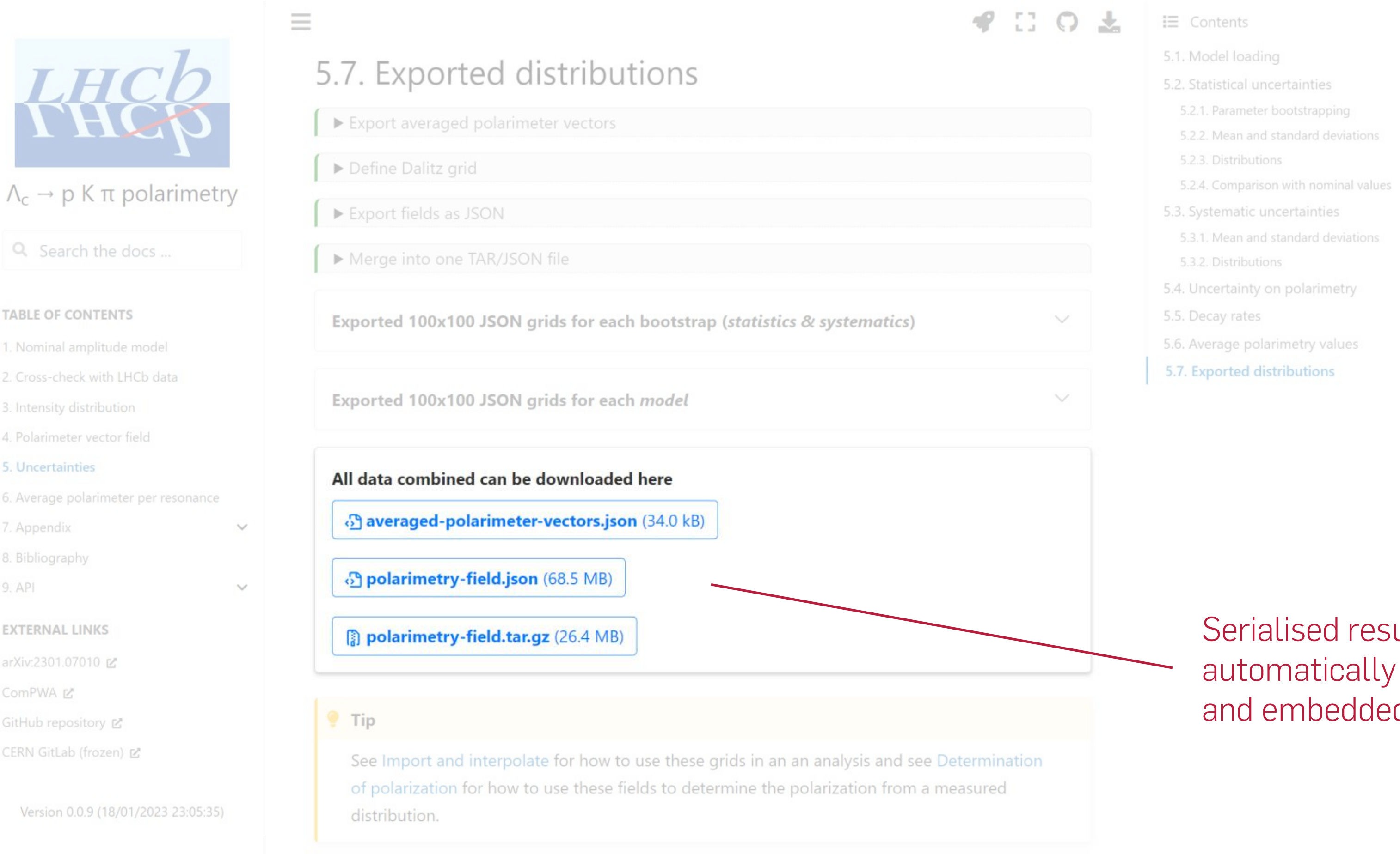
Model

High performance

Output from resource-intensive computations is rendered alongside the mathematical models

lc2pkpi-polarimetry.docs.cern.ch

Self-documenting workflow in action



Λc → p K π polarimetry

Search the docs ...

TABLE OF CONTENTS

1. Nominal amplitude model
2. Cross-check with LHCb data
3. Intensity distribution
4. Polarimeter vector field
5. Uncertainties
6. Average polarimeter per resonance
7. Appendix
8. Bibliography
9. API

EXTERNAL LINKS

- arXiv:2301.07010
- ComPWA
- GitHub repository
- CERN GitLab (frozen)

Version 0.0.9 (18/01/2023 23:05:35)

5.7. Exported distributions

▶ Export averaged polarimeter vectors

▶ Define Dalitz grid

▶ Export fields as JSON

▶ Merge into one TAR/JSON file

Exported 100x100 JSON grids for each bootstrap (*statistics & systematics*)

Exported 100x100 JSON grids for each *model*

All data combined can be downloaded here

- **averaged-polarimeter-vectors.json** (34.0 kB)
- **polarimetry-field.json** (68.5 MB)
- **polarimetry-field.tar.gz** (26.4 MB)

Tip

See Import and interpolate for how to use these grids in an an analysis and see Determination of polarization for how to use these fields to determine the polarization from a measured distribution.

Contents

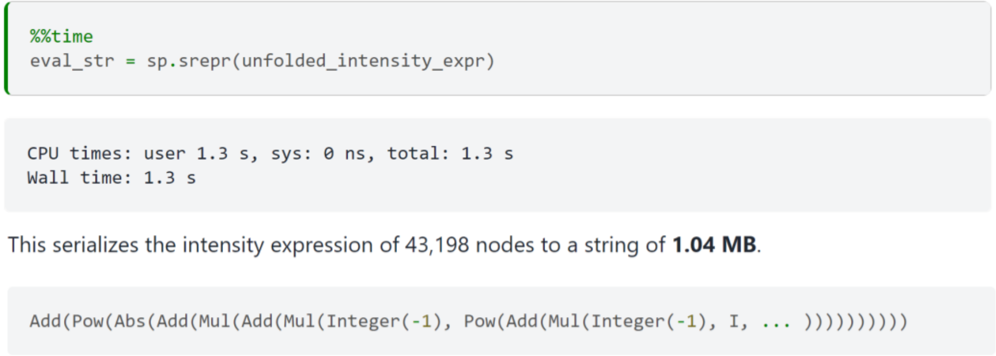
- 5.1. Model loading
- 5.2. Statistical uncertainties
  - 5.2.1. Parameter bootstrapping
  - 5.2.2. Mean and standard deviations
  - 5.2.3. Distributions
  - 5.2.4. Comparison with nominal values
- 5.3. Systematic uncertainties
  - 5.3.1. Mean and standard deviations
  - 5.3.2. Distributions
- 5.4. Uncertainty on polarimetry
- 5.5. Decay rates
- 5.6. Average polarimetry values
- 5.7. Exported distributions

Serialised results automatically exported and embedded in website compwa-org.rtfd.io/report/024.html

Serialization | Preserving CAS models

Option 1: Serialize the full tree

- SymPy's srepr() method can efficiently dump large models to disk
- Importing back is also fast enough
- Interoperability: allows importing into different SymPy versions

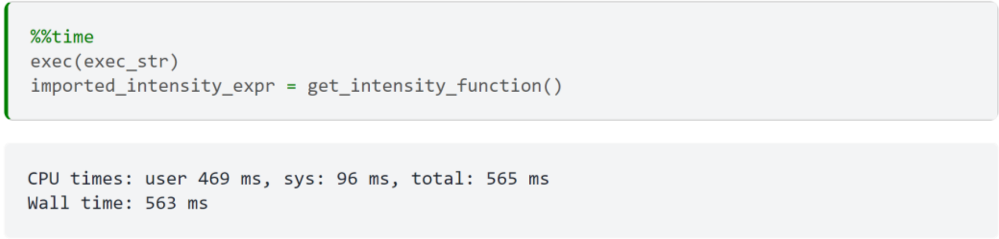


```
%%time
eval_str = sp.srepr(unfolded_intensity_expr)
```

```
CPU times: user 1.3 s, sys: 0 ns, total: 1.3 s
Wall time: 1.3 s
```

This serializes the intensity expression of 43,198 nodes to a string of **1.04 MB**.

```
Add(Pow(Abs(Add(Mul(Add(Mul(Integer(-1), Pow(Add(Mul(Integer(-1), I, ... ))))))))))
```



```
%%time
exec(exec_str)
imported_intensity_expr = get_intensity_function()
```

```
CPU times: user 469 ms, sys: 96 ms, total: 565 ms
Wall time: 563 ms
```

compwa-org.rtfd.io/report/024.html

Serialization | Preserving CAS models

Option 1: Serialize the full tree

- SymPy's srepr() method can efficiently dump large models to disk
- Importing back is also fast enough
- Interoperability: allows importing into different SymPy versions
- Also possible to serialize to **MathML**

```python
from sympy.printing.mathml import MathMLPresentationPrinter

printer = MathMLPresentationPrinter()
xml = printer._print(expr)
xml.toprettyxml().replace("\t", "  ")
```

$$-x^{2} + \frac{\sin\left(xy\right)}{2} + \frac{1}{z}$$

```xml
<mrow>
  <mrow>
    <mo>-</mo>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
  </mrow>
  <mo>+</mo>
  <mrow>
    <mfrac>
      <mrow>
        <mi>sin</mi>
        <mfenced>
          <mrow>
            <mi>x</mi>
            <mo>&InvisibleTimes;</mo>
            <mi>y</mi>
          </mrow>
        </mfenced>
      </mrow>
      <mn>2</mn>
    </mfrac>
  </mrow>
  <mo>+</mo>
  <mfrac>
    <mn>1</mn>
    <mi>z</mi>
  </mfrac>
</mrow>
```

Human-readable?

Serialization | Preserving CAS models

Option 2: Common sub-expressions

- Dynamically identify common sub-nodes in the expression tree
- Need to match expression patterns
- Does not result in familiar physics definitions

```
sub_exprs, common_expr = sp.cse(unfolded_intensity_expr)
```

$$
\begin{aligned}
I = & \left|x_{113}x_{118} + x_{205}x_{210} + x_{220}x_{223} + x_{239}x_{240} - x_{247}x_{249} - x_{256}x_{258} - x_{262}x_{263} - x_{268}x\right. \\
& + \left|-x_{113}x_{249} - x_{118}x_{247} + x_{205}x_{269} + x_{210}x_{268} - x_{220}x_{240} + x_{223}x_{239} + x_{256}x_{263} -\right. \\
& + \left|-x_{113}x_{263} + x_{118}x_{262} + x_{205}x_{223} + x_{210}x_{220} - x_{239}x_{269} + x_{240}x_{268} - x_{247}x_{258} -\right. \\
& + \left|x_{118}\left(x_{251}x_{281} + x_{253}x_{283} + x_{255}x_{285}\right) + x_{210}\left(-x_{226}x_{303} - x_{228}x_{304} - x_{230}x_{305} -\right.\right.
\end{aligned}
$$

$$
\begin{aligned}
x_0 &= m_{K(1430)}^2 \\
x_1 &= m_2^2 \\
x_2 &= m_3^2 \\
x_3 &= \frac{x_1}{2} - x_2 \\
x_4 &= i\left(\sigma_1 + x_3\right) \\
x_5 &= \frac{\Gamma_{K(1430)} m_{K(1430)} x_4 e^{-\gamma_{K(1430)}\sigma_1}}{x_0 + x_3} + \sigma_1 - x_0 \\
x_6 &= \frac{\mathcal{H}^{\text{decay}}_{K(1430),0,0}}{x_5} \\
x_7 &= m_{K(700)}^2 \\
x_8 &= \frac{\Gamma_{K(700)} m_{K(700)} x_4 e^{-\gamma_{K(700)}\sigma_1}}{x_3 + x_7} + \sigma_1 - x_7 \\
x_9 &= \frac{\mathcal{H}^{\text{decay}}_{K(700),0,0}}{x_8} \\
\ldots
\end{aligned}
$$

Serialization | Preserving CAS models

Option 3: Folded nodes

- ComPWA's `ampform` builds amplitude models with 'folded' expressions
- These might be used to extract common definitions (like custom 'PDFs', maybe HS3?)

```python
s, m1, m2 = sp.symbols("s m1 m2")
q = BreakupMomentum(s, m1, m2)
rho = PhspFactorSWave(s, m1, m2)
Math(aslatex({e: e.evaluate() for e in [rho, q]}))
```

$$
\begin{aligned}
\rho^{\text{CM}}\left(s\right) &= \frac{i \left(- \left(m_{1}^{2} - m_{2}^{2}\right) \left(- \frac{1}{\left(m_{1} + m_{2}\right)^{2}} + \frac{1}{s}\right) \log{\left(\frac{m_{1}}{m_{2}} \right)} + \frac{2 \log{\left(\frac{m_{1}^{2} + m_{2}^{2} + 2 \sqrt{s} q(s) - s}{2 m_{1} m_{2}} \right)} q(s)}{\sqrt{s}}\right)}{\pi} \\
q\left(s\right) &= \frac{\sqrt{\frac{\left(s - \left(m_{1} - m_{2}\right)^{2}\right) \left(s - \left(m_{1} + m_{2}\right)^{2}\right)}{s}}}{2}
\end{aligned}
$$

```python
from ampform.sympy import unevaluated_expression

@unevaluated_expression
class PhspFactorSWave(sp.Expr):
    s: sp.Symbol
    m1: sp.Symbol
    m2: sp.Symbol
    _latex_repr_ = R"\rho^\text{{CM}}\left({s}\right)"

    def evaluate(self) -> sp.Expr:
        s, m1, m2 = self.args
        q = BreakupMomentum(s, m1, m2)
        return 16 * sp.pi * sp.I * (
            (2 * q / sp.sqrt(s))
            * sp.log((m1**2 + m2**2 - s + 2 * sp.sqrt(s) * q) / (2 * m1 * m2))
            - (m1**2 - m2**2) * (1 / s - 1 / (m1 + m2) ** 2) * sp.log(m1 / m2)
        ) / (16 * sp.pi**2)

@unevaluated_expression
class BreakupMomentum(sp.Expr):
    s: sp.Symbol
    m1: sp.Symbol
    m2: sp.Symbol
    _latex_repr_ = R"q\left({s}\right)"

    def evaluate(self) -> sp.Expr:
        s, m1, m2 = self.args
        return sp.sqrt((s - (m1 + m2) ** 2) * (s - (m1 - m2) ** 2) / (s * 4))
```

Reproducibility | Pinning dependencies

Python does not have built-in dependency pinning

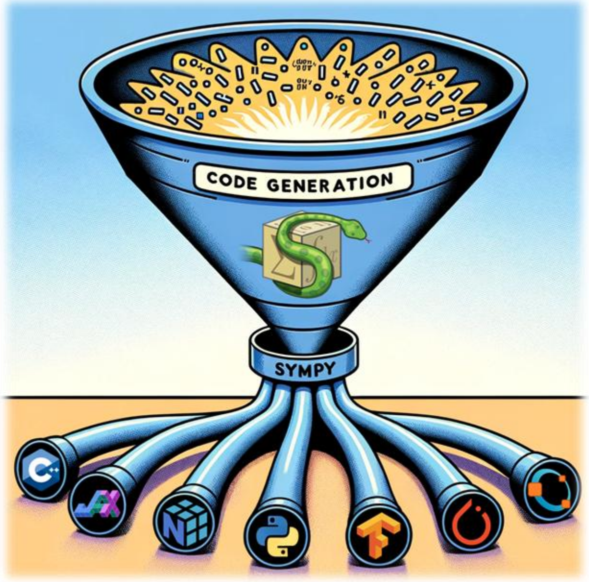
- Tools exist for pinning *all* required Python packages, e.g. pip-tools most reliable
- Only for specific Python versions
- ComPWA/update-pip-constraints used to pin analyses dependencies for *all* supported Python versions

```
.constraints/
├── py3.7.txt
├── py3.8.txt
├── py3.9.txt
├── ...
└── py3.12.txt
```

```
pip install my-package -c constraints/py3.11.txt
```

Summary

- **New techniques to separate physics from number crunching**
  - High performance with computational backends from ML
  - Flexibility and transparency with a CAS
  - Result: living documentation and self-documenting workflow
  - Bridges gap between user and developer
- **Polarimetry analysis proves that symbolic expressions:**
  - Example of self-documenting workflow with CAS
  - Analysis preserved and fully reproducible
- **Reproducibility and interoperability**
  - CAS models can easily be serialized to and from human-readable format
  - Pinning Python constraints makes analyses reproducible



Discussion

- What does human-readable mean for an amplitude model?
  Mathematics is the language we all know
- How to define interfaces for symbolic expressions? Compatibility with HS3?
- Are serialization, reproducibility and interoperability complementary?
  - Cross-checks between frameworks
  - Cross-experiment model, parameter, and data sharing
  - Extending existing analyses to other channels
- Hosting **benchmark/test amplitude analyses** and data samples
  See PyHEP.dev discussion
- Access to clusters through Jupyter notebooks for fits
  Dask seems to be a way out

Discussion

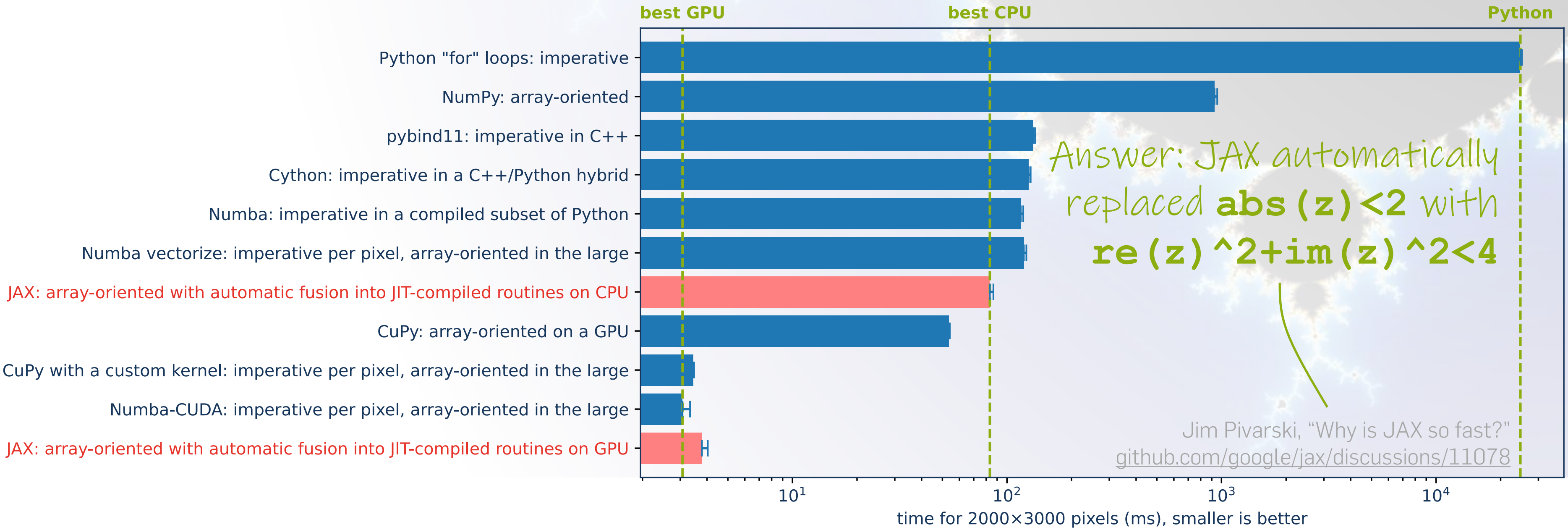
- What does human-readable mean for an amplitude model?
  Mathematics is the language we all know
- How to define interfaces for symbolic expressions? Compatibility with HS3?
- Are serialization, reproducibility and interoperability complementary?
  - Cross-checks between frameworks
  - Cross-experiment model, parameter, and data sharing
  - Extending existing analyses to other channels
- Hosting **benchmark/test amplitude analyses** and data samples
  See PyHEP.dev discussion
- Access to clusters through Jupyter notebooks for fits
  Dask seems to be a way out

Thank you for your attention!

Appendix

Array-oriented backends | Smart optimizations

Performance example: Mandelbrot set with pure C++ and different Python backends

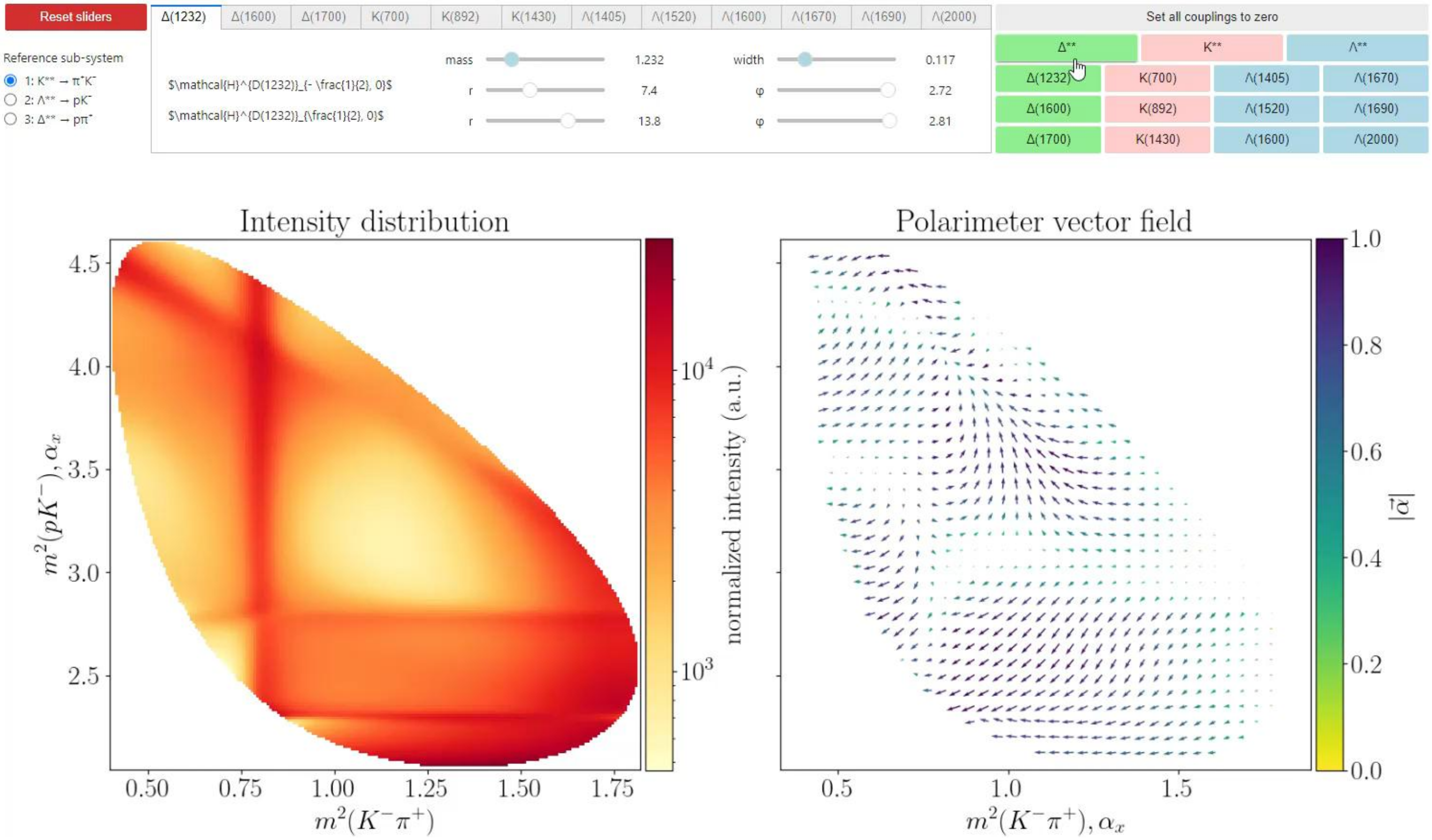


Answer: JAX automatically replaced **`abs(z)<2`** with **`re(z)^2+im(z)^2<4`**

Jim Pivarski, “Why is JAX so fast?”
github.com/google/jax/discussions/11078

lc2pkpi-polarimetry.docs.cern.ch

Performance demo | Interactive widget



Reference sub-system

● 1: $K^{**} \to \pi^+K^-$
○ 2: $\Lambda^{**} \to pK^-$
○ 3: $\Delta^{**} \to p\pi^+$

mass 1.232 width 0.117

`$\mathcal{H}^{D(1232)}_{- \frac{1}{2}, 0}$` r 7.4 φ 2.72

`$\mathcal{H}^{D(1232)}_{\frac{1}{2}, 0}$` r 13.8 φ 2.81

[link to video]

High performance
Intensity and vector field are computed upon each modification to the sliders
(recorded on laptop)

Design | Layered software development

- CAS allows us to **separate physics from number crunching**
- Symbolic expressions become a **Single Source of Truth** for physics implementations
- Model building through layers of configurability and generalization
  1. Build up symbolic models directly in a script
  2. Generalize model building with functions and classes
  3. Project evolves into generalized library
- Result: **grow a self-documenting collection of tools** for amplitude model building

Polarimetry project can also formulate models for other channels

1.

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

2.

```python
builder = ampform.get_builder(reaction)
for particle in reaction.get_intermediate_particles():
    builder.dynamics.assign(particle.name,
create_relativistic_breit_wigner)
model = builder.formulate()
```

Search the docs ...

Usage

Formulate amplitude model

Modify amplitude model

Inspect model interactively

Helicity versus canonical

Dynamics

Bibliography

API

dynamics

builder

kmatrix

helicity

sympy

kinematics

Changelog

Upcoming features

Help developing

COMPWA ORGANIZATION

Website

GitHub Repositories

About

```
class EnergyDependentWidth(s: Symbol, mass0: Symbol, gamma0: Symbol, m_a: Symbol, m_b: Symbol, angular_momentum: Symbol, meson_radius: Symbol, phsp_factor: Optional[PhaseSpaceFactorProtocol] = None, name: Optional[str] = None, evaluate: bool = False)
```

[source]

Bases: `ampform.sympy.UnevaluatedExpression`

Mass-dependent width, coupled to the pole position of the resonance.

See PDG2020, §Resonances, p.6 and [11], equation (6). Default value for `phsp_factor` is `PhaseSpaceFactor()`.

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

With that in mind, the “mass-dependent” width in a `relativistic_breit_wigner_with_ff` becomes:

$$\Gamma_0(s) = \frac{\Gamma_0 B_L^2\left(q^2(s)\right)\rho(s)}{B_L^2\left(q^2\left(m_0^2\right)\right)\rho\left(m_0^2\right)} \tag{3}$$

by (2), and ρ is (by

otocol

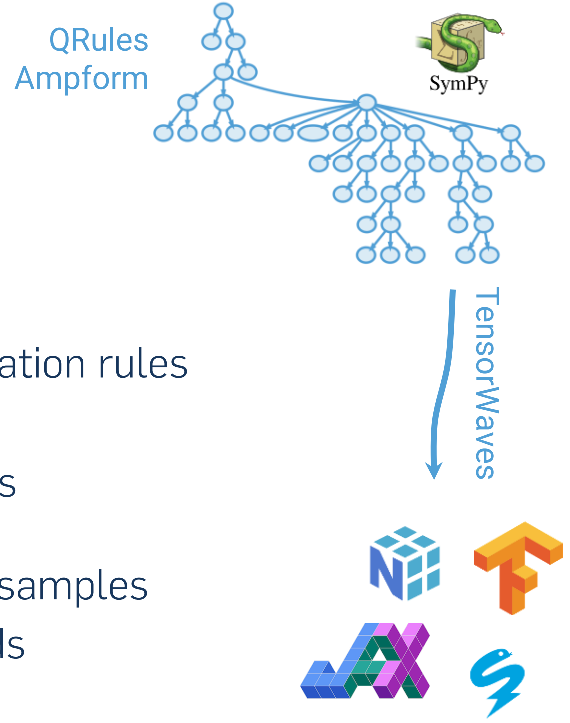
```
class PhaseSpaceFactor(s: Symbol, m_a: Symbol, m_b: Symbol, **hints: Any)
```

[source]

Standard phase-space factor, using `BreakupMomentumSquared()`.

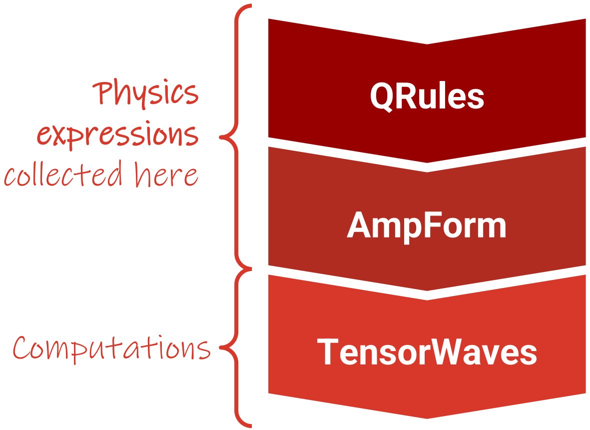
See PDG2020, §Resonances, p.4, Equation (49.8).

Design | The ComPWA project



Common Partial Wave Analysis

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



Physics expressions collected here

Computations

QRules

AmpForm

TensorWaves

Automated quantum number conservation rules

Formulate symbolic amplitude models

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