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

\[ J/\psi \to \Sigma^+ + K^0 \]
\[ J/\psi \to \Sigma^- + \bar{p} \]
\[ J/\psi \to \bar{p} + K^0 \]

Input data
3 four-momenta per collision

\[
\begin{align*}
E & \quad p_x & \quad p_y & \quad p_z \\
0.05325 & -0.102226 & -0.271504 & 0.29496 \\
1.30563 & -0.324557 & 0.223228 & 1.37042 \\
-1.35888 & 0.426783 & 0.048276 & 1.43152 \\
-0.23327 & 0.509333 & 0.499320 & 0.75844 \\
-0.68438 & 0.291036 & -0.781209 & 1.09914 \\
1.32055 & -0.258056 & -0.375920 & 1.40356 \\
-1.32055 & -0.258056 & 0.375920 & -1.40356 \\
-0.68438 & 0.291036 & 0.781209 & -1.09914 \\
1.32055 & -0.258056 & -0.375920 & 1.40356
\end{align*}
\]
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
  - need to quickly try out different parameterizations
  - fits can take several weeks
- Complicated parametrizations and estimators
- Theory is hard to get into
- Relatively small community (but growing interest!)

fast computations

flexibility
documentation

compwa@ep1.rub.de

Remco de Boer (RUB) — The ComPWA project — CHEP2023

compwa-org.rtfd.io
Amplitude analysis software

Has led to a large number of analysis packages and scripts

GPUPWA  TFPWA  Laura++

PyPWA  TensorFlowAnalysis  RooFit

AMPGEN  Pawian

PyLFit  HYDRA

PWA frameworks

Scripts using fitter packages

compwa@ep1.rub.de  compwa-org.rtfd.io
Amplitude analysis software

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
Differentiable programming

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

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

```python
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
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

Usually all that the user needs to do
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)
```

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)
```

**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
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

```matlab
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).*s^0.01))./(4*m0.^2)).*(s - 0.25).*sqrt(m0.*2)./(s^3/2).*sqrt((m0.*2 - 0.01).*(s - 0.25).*s^-0.25).*s^1.
end
```

SymPy 'lambdification'
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 - & 0.01d0)*SQRT(m0**2)/(s**((3.0d0/2.0d0)**2)*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
```
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

```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 - 0.01)*sqrt(pow(m0, 2) + pow(m0, 2) - (s - 0.25)*pow(m0, 2) - 0.25)*pow(m0, 2) - 0.01)*pow(m0, 2) - 0.01) + pow(m0, 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

```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'
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

Physics separated from the ‘number crunching’
Works just as well for models with tens of thousands of nodes
The **ComPWA project**

**Common Partial Wave Analysis**

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

- **QRules**
  - Automated quantum number conservation rules

- **AmpForm**
  - Formulate symbolic amplitude models

- **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

compwa@ep1.rub.de
compwa-org.rtfd.io
**Living documentation**

**Codebase**

```python
@implement_doi_method
class EnergyDependentWidth(UnevaluatedExpression):
    r"""Mass-dependent width, coupled to the pole position of the resonance.
    See :code:`p Raei-rad: B; 2020; Resonances; p.6` and :
cite'Banerji, D. and Prabhakaran, S., 2006, equation (6). Default value for :
code:`phsp_factor` is :math:`\times PhaseSpaceFactor`.
    Note that the `Blatt-WeisskopfSquared` of AmpForm is normalized in the sense that equal powers of :math:`m` appear in the nominator and the denominator, while the definition in the PDG (as well as some other sources), always have :math:`m^2` in the nominator of the Blatt-Weisskopf. In that case, one needs an additional factor :math:`\left(\alpha/\sqrt{2}\right)^2` in the definition of `BlattWeisskopfSquared`
    ...

    def evaluate(self) -> op.Expr:
        s, m, m0, gamma0, m_a, m_b, angular_momentum, meson_radius = self.args
        q2_squared = BreakupMomentumSquared(s, m_a, m_b)
        q0_squared = BreakupMomentumSquared(mass0**2, m_a, m_b)
        form_factor_sq = BlattWeisskopfSquared(
            angular_momentum,
            zeta_squared = meson_radius**2,
        )
        form_factor0_sq = BlattWeisskopfSquared(
            angular_momentum,
            zeta0_squared = meson_radius0**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(determining_indices(width))
    name = rf'\Gamma^{\alpha}_{\Gamma^0}(s)' if self.name is None else self.name
    return rf'\text{\texttt{CompWa}}{subscript}'
```

**Launch interactive examples**

**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 \( R_i \):\(^4\)

\[
K_{ij} = \sum_R \frac{g_R j_i R_j}{m_R^2 - s} + c_{ij} \tag{14}
\]

\[
\dot{K}_{ij} = \sum_R \frac{\gamma_{R_i}(s) g_{R_j}}{(m_R^2 - s)^{3/2}} \frac{g_{R_j}}{\sqrt{p_R^2}} \tag{15}
\]

\( \dot{K}_{ij} \) with \( c_{ij} \) are related to the partial wave \( \gamma_R \) of the residue functions. The result of \( K_{ij} \) in a fixed channel \( j \) is proportional to the partial wave \( \Gamma_R \) of the resonance.

**Jupyter notebooks**

- **Dynamic code allows for interactivity**
- **Can be rendered as web pages**\(^4\) Eqns. (75-78)
- **Serves as integration test**

with \( \Gamma_{R_i} \) some real constants and \( \Gamma_{R_i}^u \) the partial width of each pole. In the Lorentz-invariant form, the fixed width \( \Gamma^0 \) is replaced by an "energy dependent"

\[
\text{CoupledWidth}(\Gamma)(s) \tag{5}\]

The width for each pole can be computed as \( \Gamma_{R_i}^0 = \sum_j \Gamma_{R_{ij}}^0 \).

The production vector \( P \) is commonly parameterized
Living documentation

Codebase

![Jupyter notebooks]

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

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

    :code:`phsp_factor` is :meth:`PhaseSpaceFactor`.

    Note that the `BlattWeisskopfSquared` of AmpForm is normalized in the sense
    that equal powers of :math:`\hbar` appear in the numerator 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(\left(q/q_0\right)^2+\right)`
in the definition for `BlattWeisskopfSquared`.

    .. codeblock:: python

    ```python
    def evaluate(self) -> np.:
        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_m_squared * meson_radius**2,
        )
        form_factor_0_sq = BlattWeisskopfSquared(
            angular_momentum,
            z_m0_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_factor_0_sq) * (rho / rho0)
    ```

    def latex(self, printer: LatexPrinter, *args) -> str:
        s, width, _ = self.args
        s = printer.print(s)
        subscribe = indices_to_subscribe(determining_indices(width))
        name = RF''/ Gamma subscrip\(,\)` if self_name is None else self_name
        return name = RF''|a|''[\(,\)]
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

```bash
pip install qrules
pip install ampform
pip install tensorwaves
```
**Core:** ‘search engine’ for quantum numbers

Get particle properties:

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

Find particles by quantum number:

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

*(PDG info computed from the scikit-hep `particle` package)*

Check which conservation rules are violated:

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

*(frozenset({"c_parity_conservation"}))*

Also a library of conservation rules

*Also a library of 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**

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

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

**Selects conservation rules**

- $J/psi(1S)\rightarrow K(1(1650))\rightarrow 0[0]$  
  $l=(2, 0)\quad s=(1, 0)\quad P=+1$

- $J/psi(1S)\rightarrow K(1(1820))\rightarrow 0[0]$  
  $l=(1, 0)\quad s=(2, 0)\quad P=+1$

- $J/psi(1S)\rightarrow K(1(1675))\rightarrow 0[2]$  
  $l=(1, 0)\quad s=(0, 0)\quad P=-1$

- $J/psi(1S)\rightarrow N(1(1720))\rightarrow 0[2]$  
  $l=(2, 0)\quad s=(1, 0)\quad P=-1$

- $J/psi(1S)\rightarrow N(1(1720))\rightarrow 0[2]$  
  $l=(1, 0)\quad s=(1, 0)\quad P=+1$

- $J/psi(1S)\rightarrow N(1(1720))\rightarrow 0[2]$  
  $l=(1, 0)\quad s=(1, 0)\quad P=+1$

- $J/psi(1S)\rightarrow N(1(1720))\rightarrow 0[2]$  
  $l=(1, 0)\quad s=(1, 0)\quad P=+1$

- $J/psi(1S)\rightarrow N(1(1720))\rightarrow 0[2]$  
  $l=(1, 0)\quad s=(1, 0)\quad P=+1$

- $J/psi(1S)\rightarrow N(1(1720))\rightarrow 0[2]$  
  $l=(1, 0)\quad s=(1, 0)\quad P=+1$
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

\[
\rho(s) \sum_{R=1}^{n_R} \frac{\Gamma(s)\gamma_R^2}{-s+m_R^2}
- i\rho(s) \sum_{R=1}^{n_R} \frac{\Gamma(s)\gamma_R^2}{-s+m_R^2} + 1
\]

\[
\frac{\Gamma_{1,0}\gamma_{1,0}^2 m_1}{s + i\Gamma_{1,0}\gamma_{1,0}^2 m_1 + i\Gamma_{1,1}\gamma_{1,1}^2 m_1 - m_1^2}
\]
**Example:** amplitude model for $D^0 \rightarrow K^0 K^- K^*$ 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()
```

$$A_{D^0 \rightarrow K^0\phi(1020)_0;\phi(1020)_0 \rightarrow K^+_0 K^-_0} + A_{D^0 \rightarrow K^0 a_0(1450)_0; a_0(1450)_0 \rightarrow K^+_0 K^-_0} + A_{D^0 \rightarrow K^0 a_0(980)_0; a_0(980)_0 \rightarrow K^+_0 K^-_0}$$
Example: amplitude model for $D^0 \rightarrow K^0 K^- K^+$ with 3 resonances

```
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()
```

Each amplitude can be further inspected:

```
model.components[
R"A_{D^0 \rightarrow K^0 a(1450)0; a(1450)0 \rightarrow K^+ K^-} + A_{D^0 \rightarrow a(1450)0 a(1450)0 \rightarrow K^+ K^-} + A_{D^0 \rightarrow a(980)0 a(980)0 \rightarrow K^+ K^-} \]^2
```

```
C_{D^0 \rightarrow K^0 \phi(1020)0; \phi(1020)0 \rightarrow K^+ K^-} \Gamma \phi(1020) m_{\phi(1020)} \sqrt{B_1^2 (q_{12}^2 m_{\phi(1020)}^2)} D_{0,0}(-\phi_0, \theta_0, 0) D_{1,0}(-\phi_{12}^{12}, 0) D_{0,0}(-\phi_{12}^{\phi(1020), \Gamma \phi(1020) m_{\phi(1020)}})
```

```
-m_{12}^2 + m_{\phi(1020)}^2 - im_{\phi(1020) \Gamma \phi(1020) m_{\phi(1020)}}
```
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, ...)

```python
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)
```
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

![Graphs showing negative log likelihood and parameter values](image-url)
Amplitude model for $\Lambda_c \rightarrow p\pi K$
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
→ computation time decreases by 25%
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