The ComPWA project

Speeding up differentiable programming with a Computer Algebra System

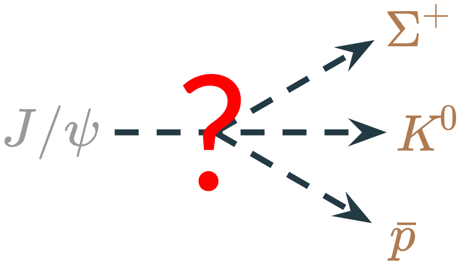
Remco de Boer
Ruhr University Bochum

15 September 2022
PyHEP Notebook Talk

launch binder

Context: **Amplitude analysis**

Aim: **study of intermediate hadronic states**

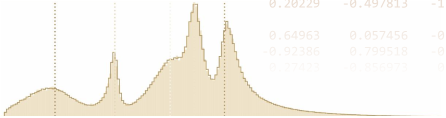


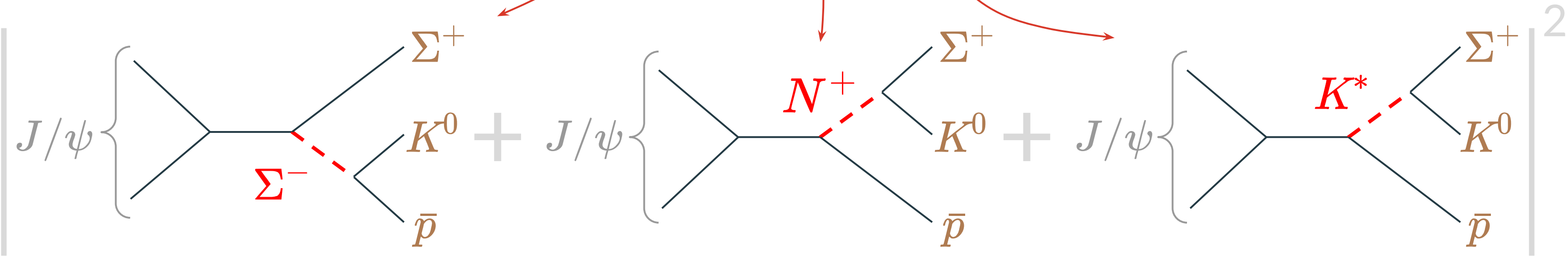
*Find **models** that correctly describe the observed **data distributions***

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





Context: **Amplitude analysis**

Very large model descriptions…

→ *flexibility*

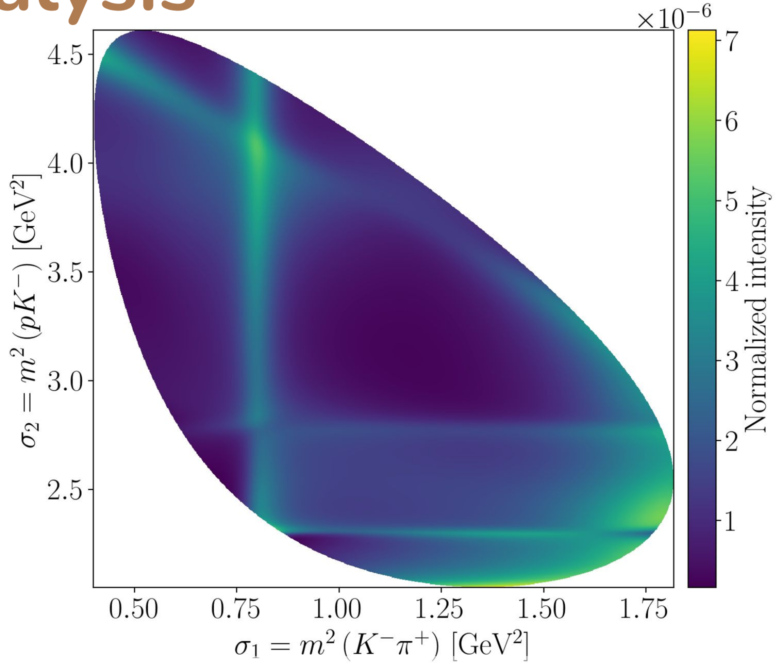
$$\sum_{\lambda_0'=-1/2}^{1/2}\sum_{\lambda_1'=-1/2}^{1/2} A^1_{\lambda_0',\lambda_1'} d^{\frac{1}{2}}_{\lambda_1',\lambda_1}\left(\zeta^1_{1(1)}\right) d^{\frac{1}{2}}_{\lambda_0,\lambda_0'}\left(\zeta^0_{1(1)}\right) + A^2_{\lambda_0',\lambda_1'} d^{\frac{1}{2}}_{\lambda_1',\lambda_1}$$

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

$$\zeta^1_{2(1)} = \operatorname{acos}\left(\frac{2m_1^2(-m_0^2-m_3^2+\sigma_3)+(m_0^2+m_1^2-\sigma_1)(-m_1^2-m_3^2+\sigma_2)}{\sqrt{\lambda(m_0^2,m_1^2,\sigma_1)}\sqrt{\lambda(\sigma_2,m_1^2,m_3^2)}}\right)$$

$$\zeta^0_{3(1)} = \operatorname{acos}\left(\frac{-2m_0^2(-m_1^2-m_3^2+\sigma_2)+(m_0^2+m_1^2-\sigma_1)(m_0^2+m_3^2-\sigma_3)}{\sqrt{\lambda(m_0^2,m_1^2,\sigma_1)}\sqrt{\lambda(m_0^2,\sigma_3,m_3^2)}}\right)$$

$$\zeta^1_{3(1)} = -\operatorname{acos}\left(\frac{2m_1^2(-m_0^2-m_2^2+\sigma_2)+(m_0^2+m_1^2-\sigma_1)(-m_1^2-m_2^2+\sigma_3)}{\sqrt{\lambda(m_0^2,m_1^2,\sigma_1)}\sqrt{\lambda(\sigma_3,m_1^2,m_2^2)}}\right)$$



$\mathcal{R}(s) =$

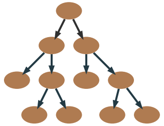
…and large, unbinned data samples

→ *performance*

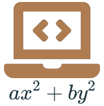
Context: **Amplitude analysis**

fast computations on large data samples

How to bring code closer to theory?



Computational back-ends from ML and data science (differentiable programming)



Formulate models with a **Computer Algebra System**

Automatically document what you implemented

flexibility to quickly formulate large models

Differentiable programming

Since a few years, 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

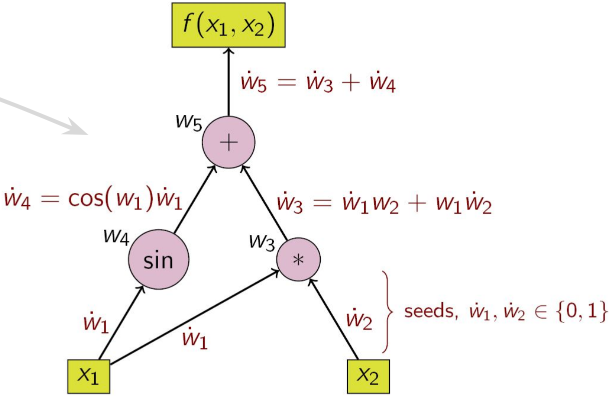
```
for (i = 0; i < rows; i++): {
  for (j = 0; j < columns; j++): {
    c[i][j] = a[i][j]*b[i][j];
  }
}
```

These backends already take a lot of work off our shoulders through:

- 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
```

Heavy lifting by optimized backend

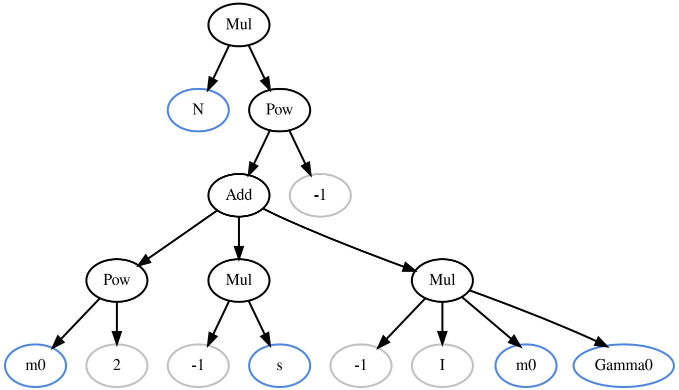


NVIDIA CUDA

Computer Algebra System

$ax^2 + by^2$

Can we make it even easier to formulate large models?



SymPy

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

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

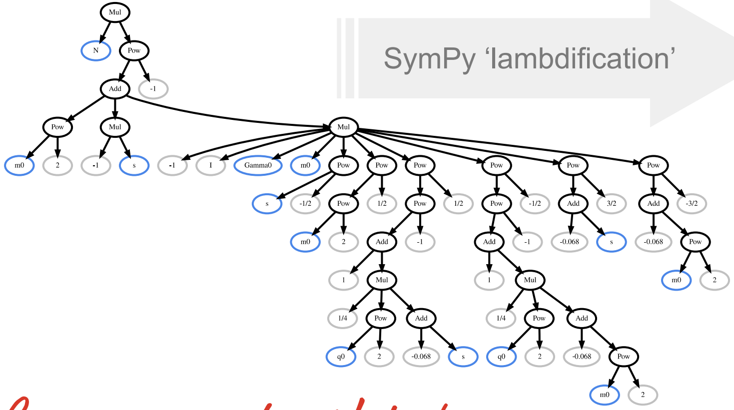
*Can serve as **template** to computational back-ends!*

CAS represents expression as a tree

Computer Algebra System

$ax^2 + by^2$

Can we make it even easier to formulate large models?



SymPy 'lambdification'

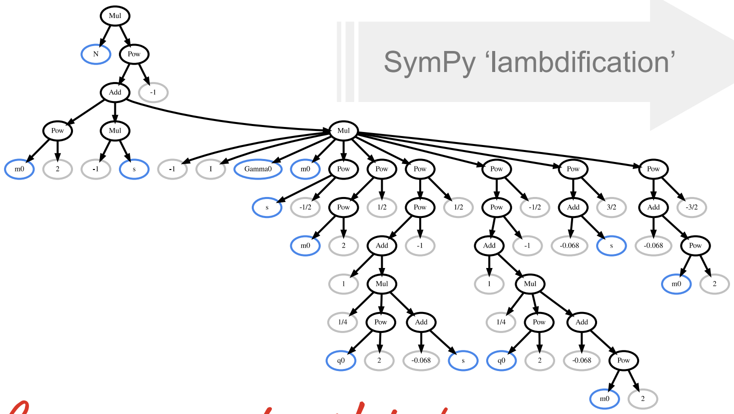
Fortran

```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
```

Can serve as ***template*** *to computational back-ends!*

Computer Algebra System

Can we make it even easier to formulate large models?



SymPy ‘lambdification’

C++

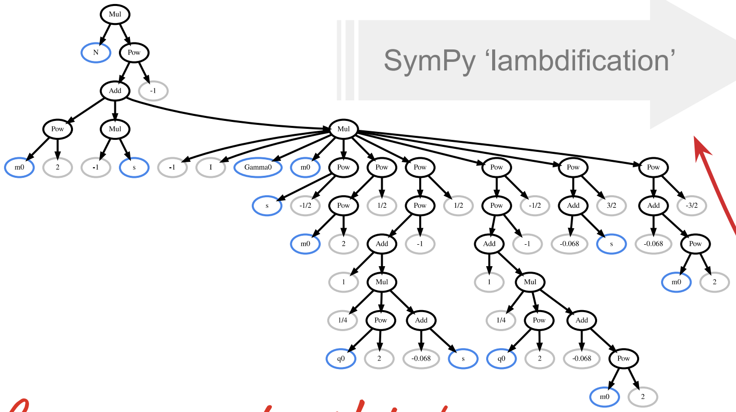
```
// my_expr.h
#ifndef PROJECT__MY_EXPR__H
#define PROJECT__MY_EXPR__H
double my_expr(double Gamma0, double N, double m0, double s);
#endif

// 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;
  my_expr_result = 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);
  return my_expr_result;
}
```

Can serve as ***template*** *to computational back-ends!*

Computer Algebra System

$ax^2 + by^2$

Can we make it even easier to formulate large models?



SymPy 'lambdification'

Python with JAX

```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
    )
```

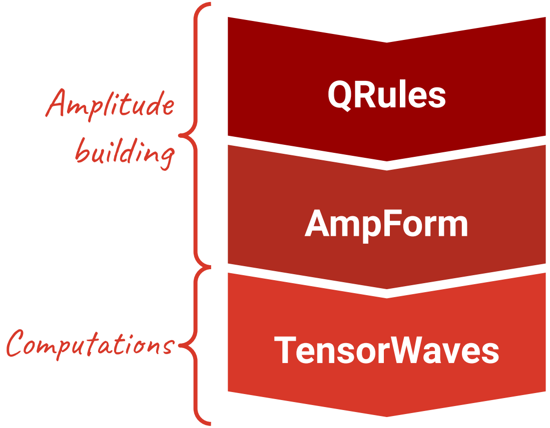
*Can serve as **template** to computational back-ends!*

Any CAS simplifications optimize the back-end code!

The ComPWA project

Common Partial Wave Analysis

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



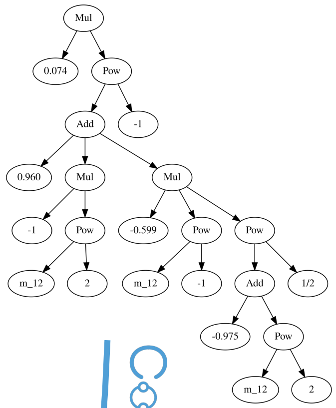
Automated quantum number conservation rules

Formulate symbolic amplitude model

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 compwa-org.rtfd.io

The ComPWA project



ComPWA

Main responsibilities:

- Standardize and automate amplitude analysis theory in CAS code
- Streamline and improve conversion from CAS to back-end
- Generate amplitude-based Monte Carlo samples
- Perform fits with different optimizers (Minuit2, SciPy, …)

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)
```

NumPy

Numba

The ComPWA project

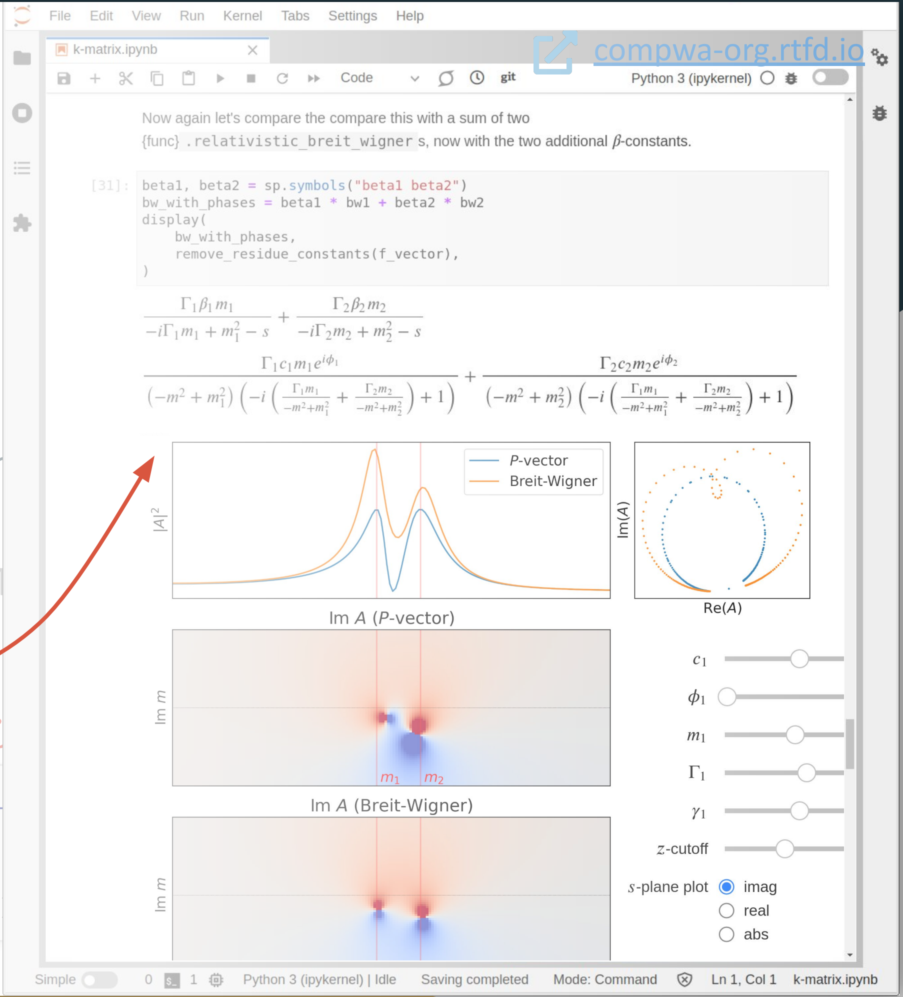
Main responsibilities:

- Standardize and automate amplitude an
- Streamline and improve conversion from
- Generate amplitude-based Monte Carlo
- Perform fits with different optimizers (M

Self-documenting workflow

Any s

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

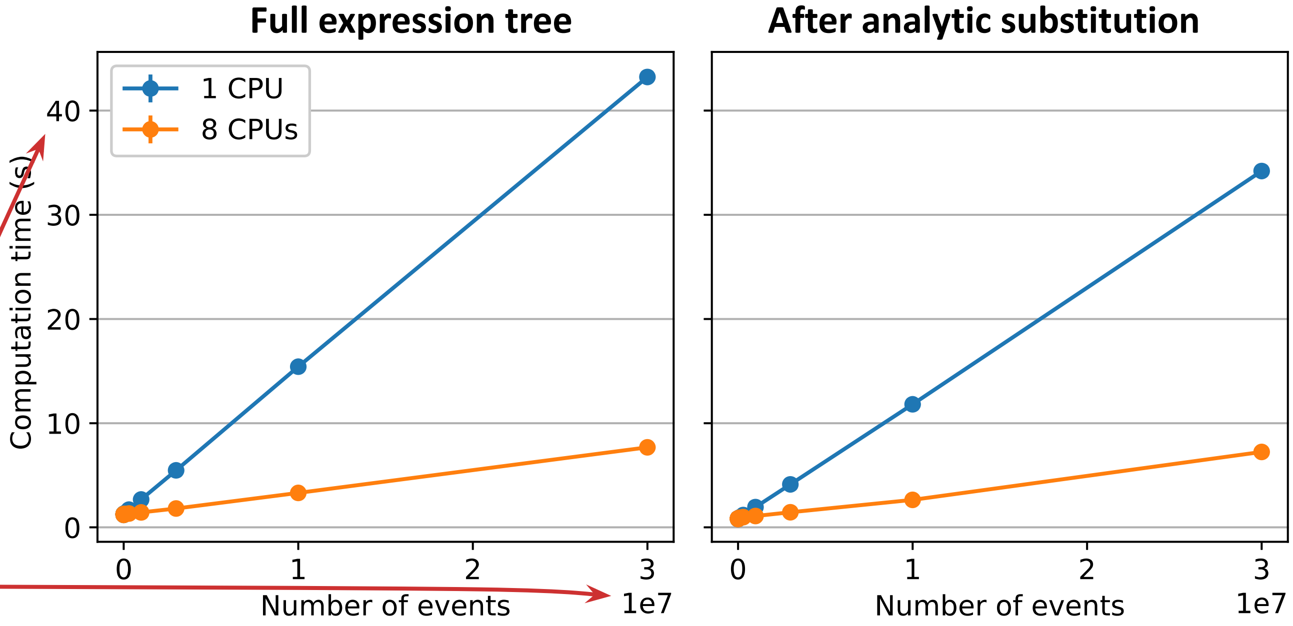


The ComPWA project

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

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

Backend: JAX



Great performance for a single fit iteration!

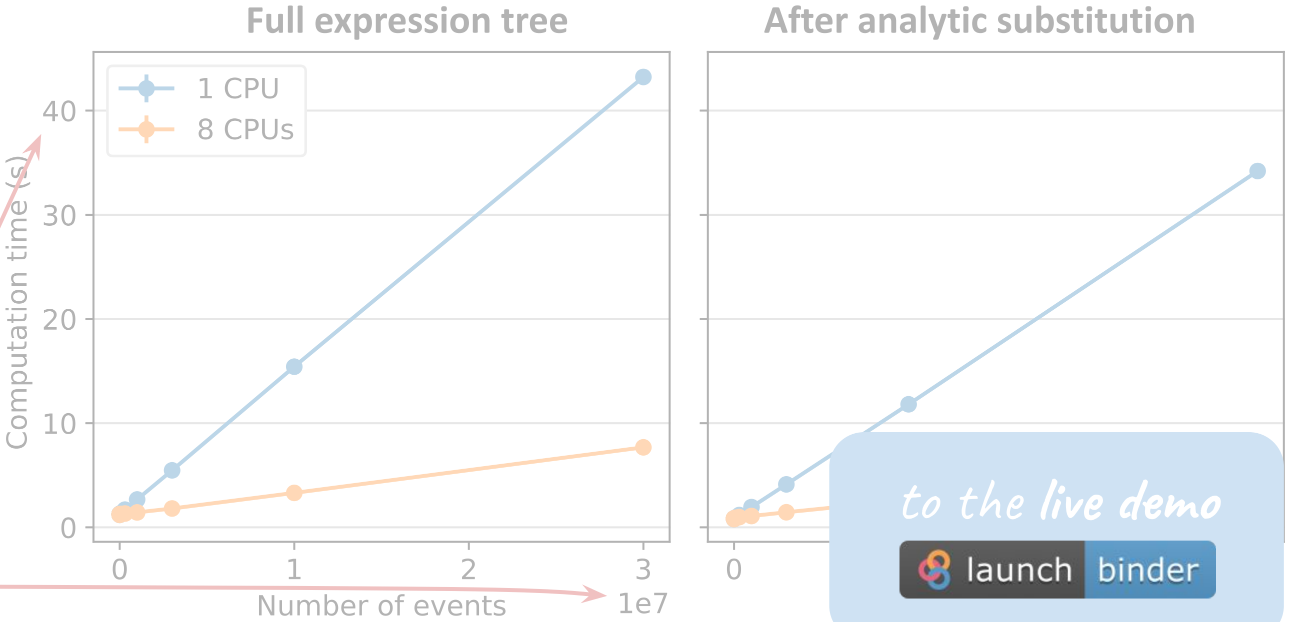
compwa-org.rtfd.io

The ComPWA project

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

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

Backend: JAX



Great performance for a single fit iteration!

to the live demo
launch binder