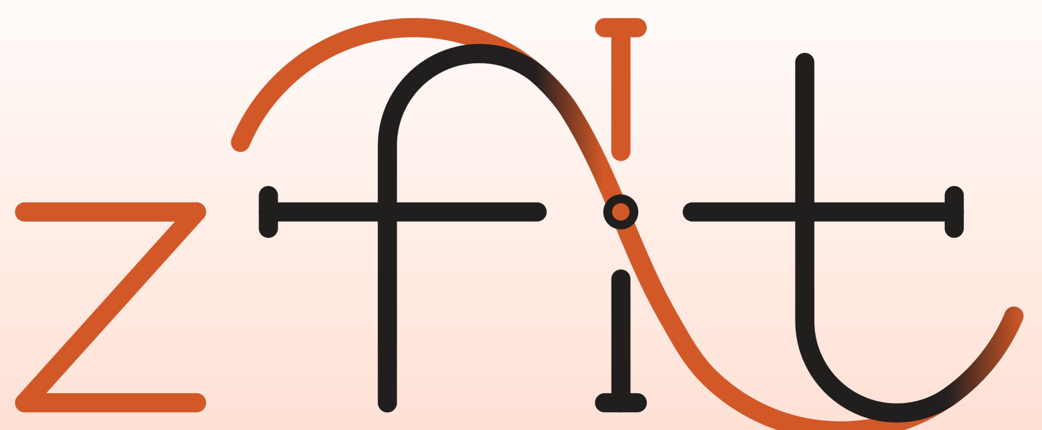
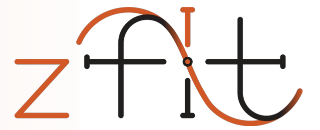


scalable pythonic fitting

Jonas Eschle on behalf of zfit
jonas.eschle@cern.ch

Before we start...

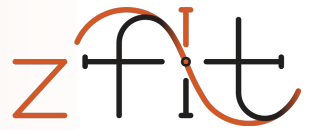


You can try out zfit interactively

https://zfit-tutorials.readthedocs.io/en/latest/

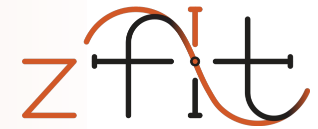
New binned fits (WIP)

A brief history



- A few years ago: analyses transition from C++ to Python
  - Scikit-HEP was created
  - Change of philosophy: non-monolithic packages
- Fitting packages still in in C++
  - Many scattered, specialized packages
  - Speed crucial aspect (and non-trivial in python)

Fitting in Python

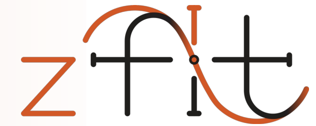


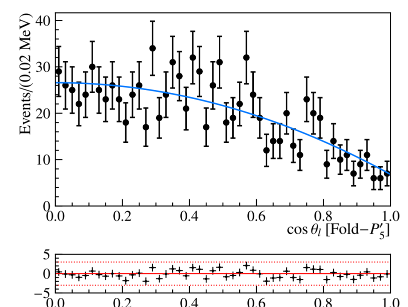
A lot of projects are around

- ~~RooFit~~
- ~~HEP Python~~
- ~~Non-HEP~~

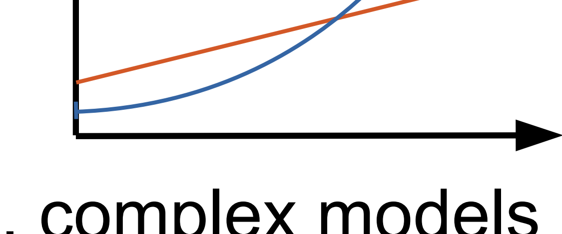
No real model fitting ecosystem/library for HEP that is well integrated into Python

HEP Model Fitting in Python





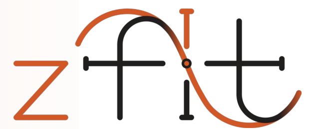
HEP
advanced features,
simply extendable

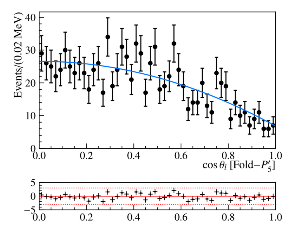


Scalable
large data, complex models

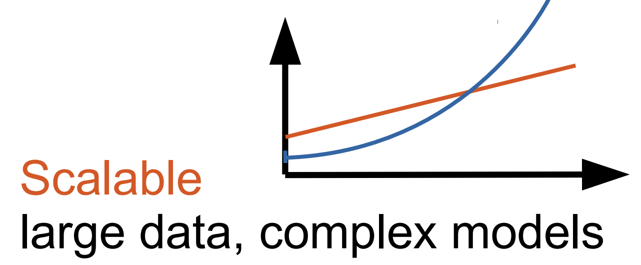
Pythonic NumPy python™
integrate into ecosystem, stable API

HEP Model Fitting in Python





HEP
advanced features,
simply extendable



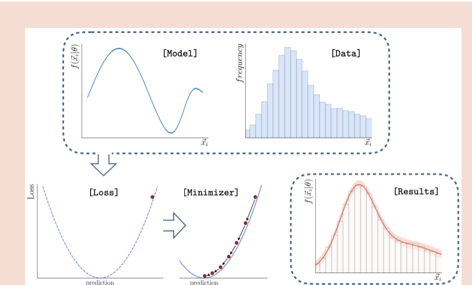
Scalable
large data, complex models

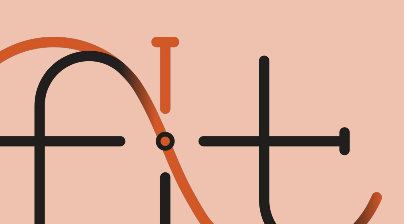
Computing backend

Pythonic

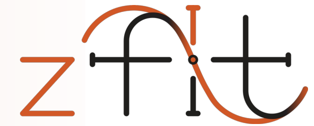
integrate into ecosystem, stable API

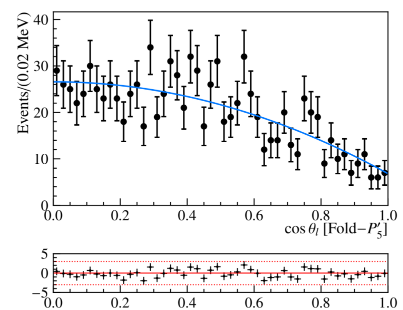
API & Workflow



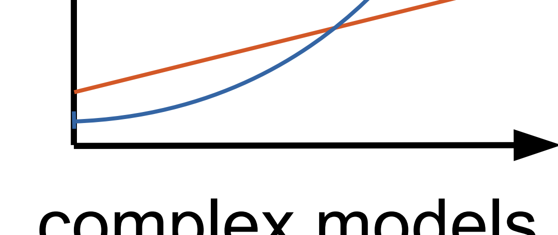


HEP Model Fitting in Python





HEP
advanced features,
simply extendable



Scalable
large data, complex models

Pythonic NumPy python™
integrate into ecosystem, stable API

Ecosystem

vector

phasespace

zfit-physics

For user contributions

Scikit HEP

Extend

hepstats

zfit-amplitude

Build on top

zfit

stable core package

TensorFlow

uproot

python™

iminuit

Boost histogram

Scikit HEP

Ecosystem

vector

phasespace

zfit-physics

For user contributions

Scikit HEP

Extend

hepstats

zfit-amplitude

Build on top

zfit

stable core package

iminuit

TensorFlow

uproot

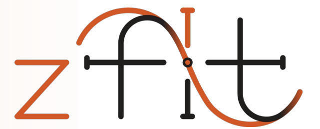
python™

Boost histogram

Scikit HEP

model fitting & sampling

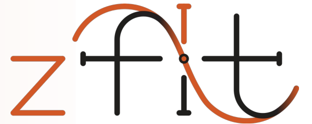
API & Workflow



Five maximally independent parts

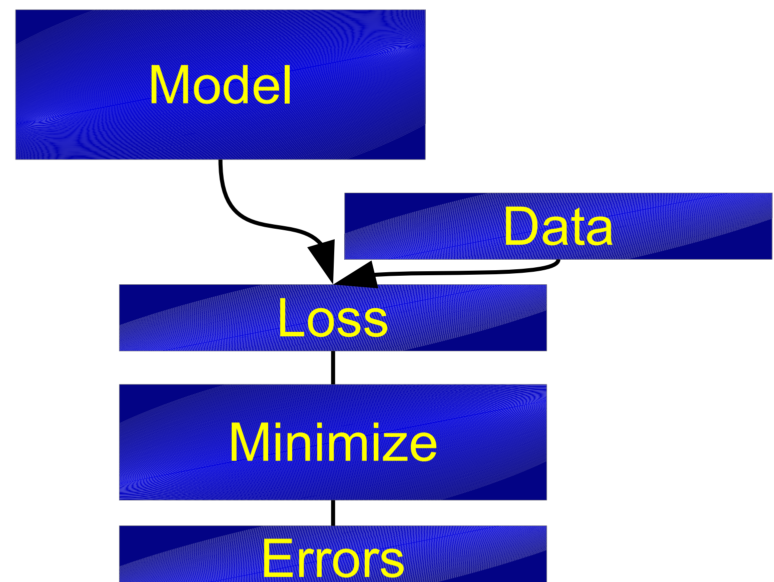
"Fits look always
 the same"

Complete fit

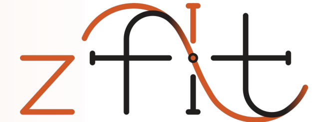


Disclaimer:
unbinned fits way more developed, binned very new in pre-release

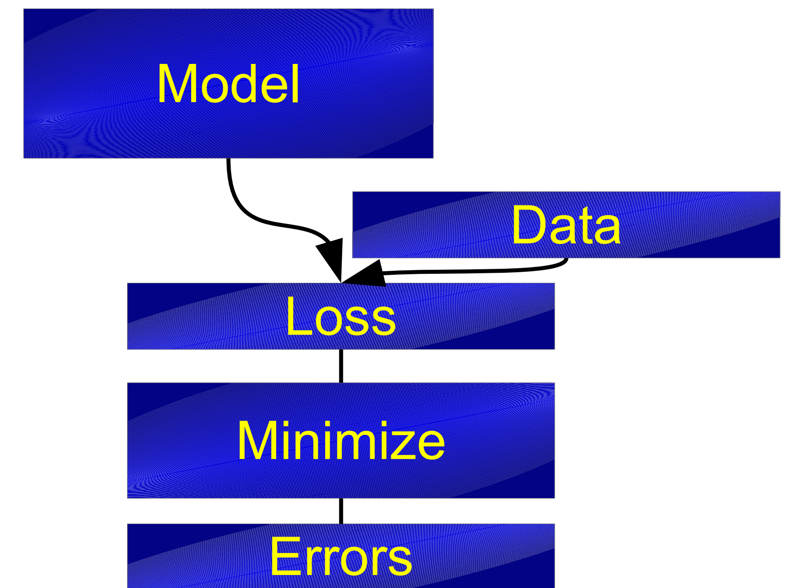
```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



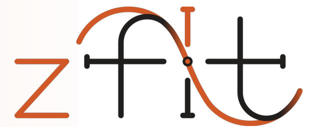
Complete fit



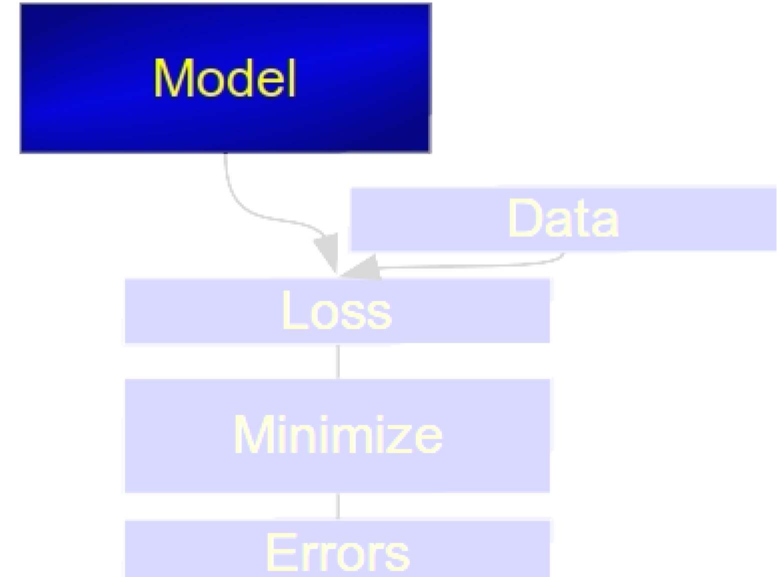

```python
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



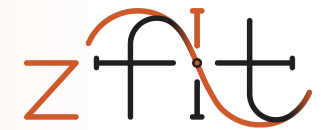
Complete fit: Model



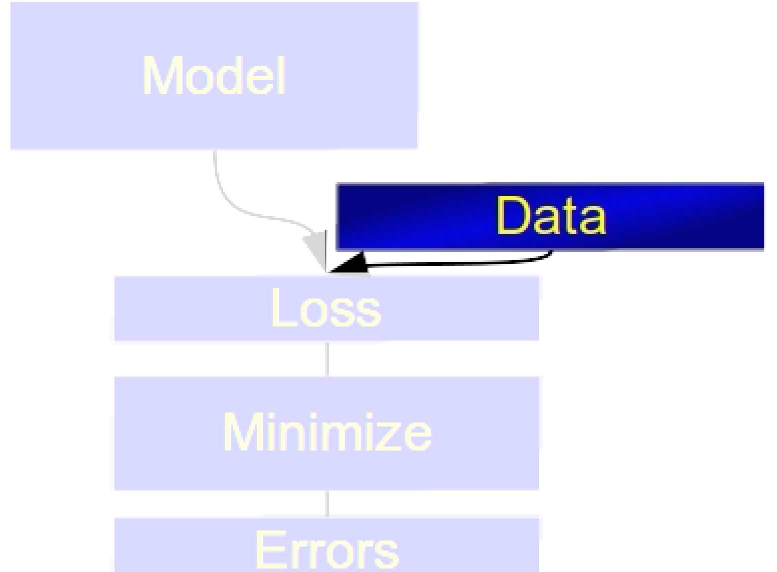
```python
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



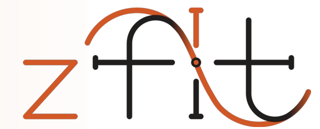
Complete fit: Data



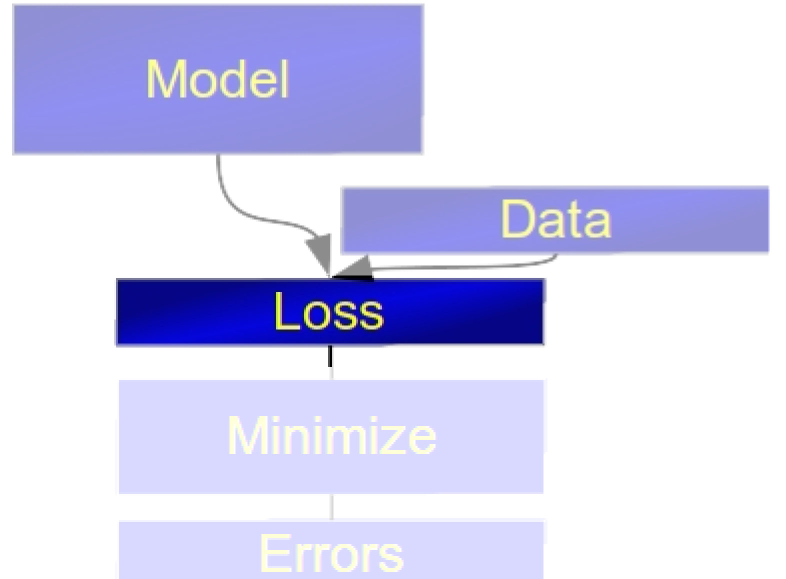
```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



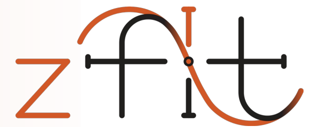
Complete fit: Loss



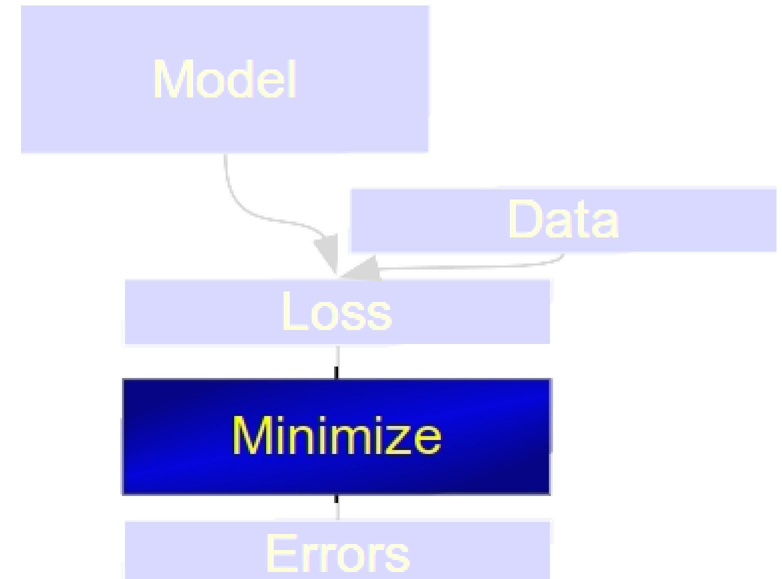

```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



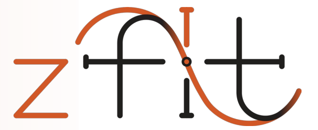
Complete fit: Minimization



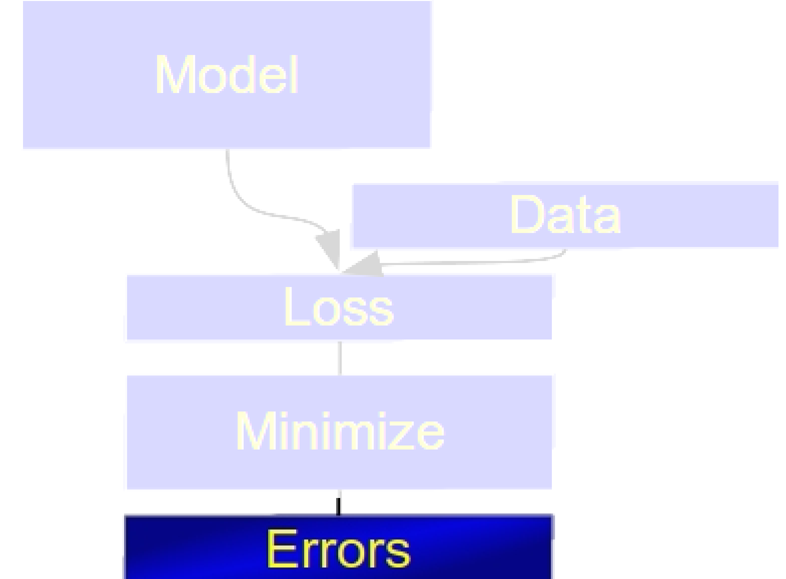

```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



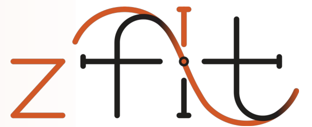
Complete fit: Result



```python
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



Basic API example



```python
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
```

```python
data = zfit.Data.from_numpy(obs=obs, array=normal_np)
```

```python
nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)
```

```python
minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)
```

```python
param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```

Model

Data

Loss

Minimize

Errors

Basic API example

Going binned

```
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)

gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
obs_binned = obs.with_binning(30)
gauss_binned = zfit.pdf.BinnedFromUnbinnedPDF(gauss, obs_binned)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)
data_binned = data.to_binned(obs_binned)

nll = zfit.loss.BinnedNLL(model=gauss_binned, data=data_binned)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```

Model

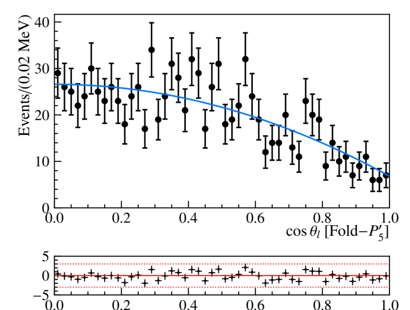
Data

Loss

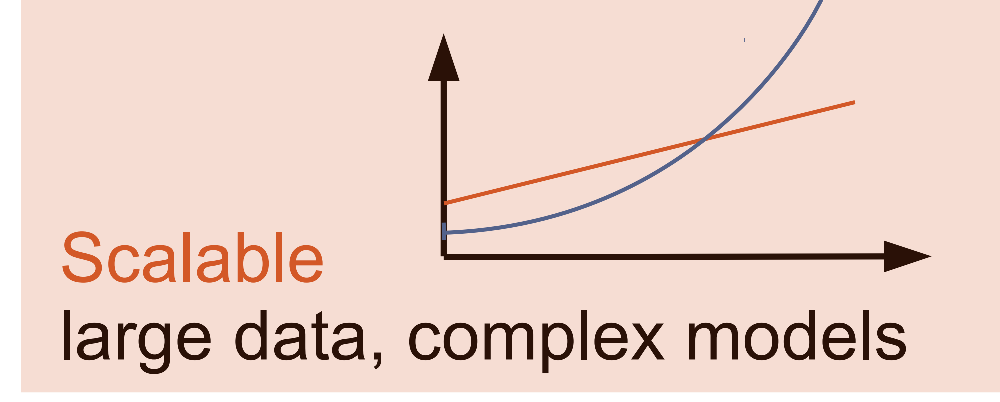
Minimize

Errors

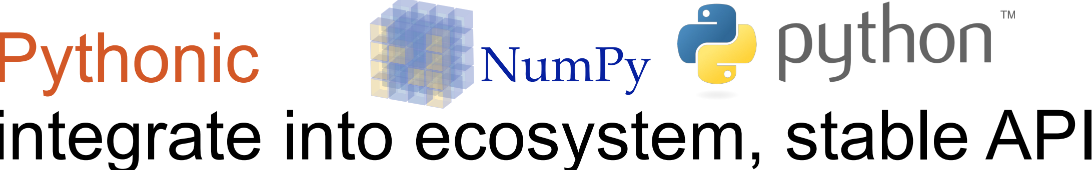
HEP Model Fitting in Python



HEP
advanced features,
simply extendable

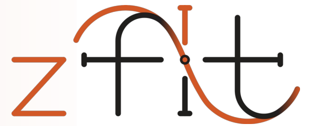


Scalable
large data, complex models



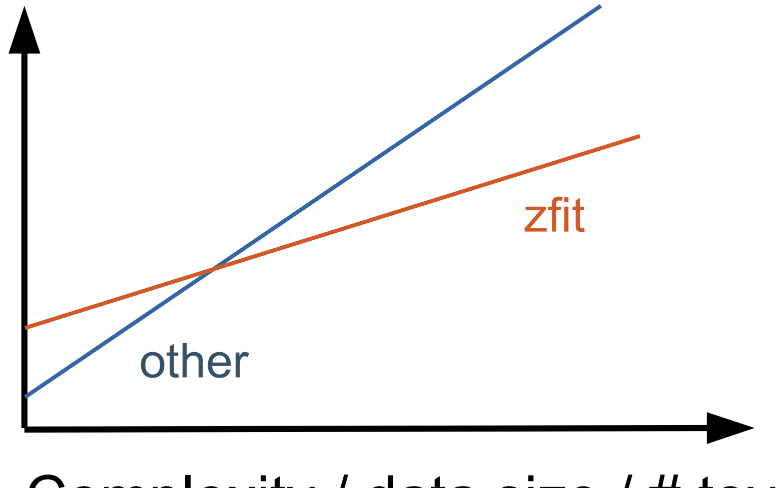
Pythonic
integrate into ecosystem, stable API

Scalable: Performance



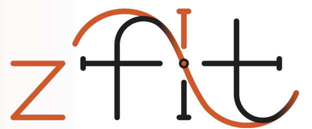
There is no free lunch

- Initial overhead, flat increase
- TensorFlow (JAX, …) backend
- JIT compiled, CPU or GPU



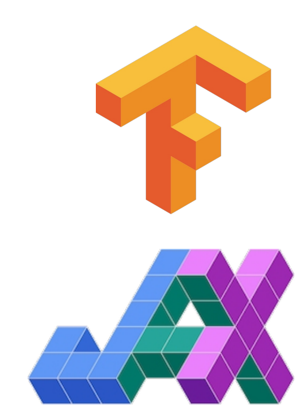
- Single, simple fit "slow"
  - 0.01 or 1 sec not relevant
  - 1 or 10 hours relevant

Backend: tracing and autograd



Tracing
execute Python once, remember ("algebraic") computation

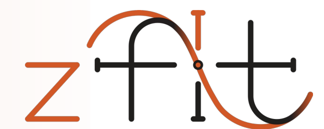
Autograd
automatic gradient of function



Recent rise of big data industry created libraries that support this

Includes GPU support, optimizations, caching,...

Delegating the workload



	C++ library	Numpy based	zfit
HEP specific content/API			
Models		SciPy	TF Probability
Gradients			
Computational optimizations			
Parallelization/GPU		Numba NumPy	TensorFlow intel NVIDIA
Low level handling		python	python

Main backend: TensorFlow

- By Google, highly popular (150k★, top on GitHub)
- Consists of "two parts":
  - High level API for building neural networks *(NOT used!)*
  - Low level API with Numpy-style syntax

    tf.sqrt, tf.random.uniform,… or *tnp.sqrt, tnp.array, tnp.linspace*
- Two modes:
  - "numpy"-like (full Python flexibility)
  - "compiled" (very performant)

} GPU/Multi CPU support

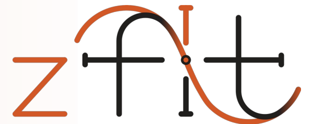
Main backend: TensorFlow

- By Google, highly popular (150k★, top on GitHub)
- Consists of "two parts":
  - High level API for building neural networks *(NOT used!)*
  - Low level API with Numpy-style syntax

    tf.sqrt, tf.random.uniform,… or *tnp.sqrt, tnp.array, tnp.linspace*
- Two modes:
  - "numpy"-like (full Python flexibility)
  - "compiled" (very performant)

  } GPU/Multi CPU support
- Supports unknown shapes (e.g. JAX does not)

What happens *exactly*?



1) Function is called with a *signature* (Tensors, Python objects)
2) Function is traced:
   - Python code is executed
   - TF code is remembered in graph
   - Caches graph with signature
3) Execute graph with inputs

```
@tf.function(autograph=False)
def add_mult(a, b, c, d):
    print("compiling...")
    tf.print("running")
    sum_ab = a + b
    sum_cd = c + d
    return sum_ab * sum_cd
```

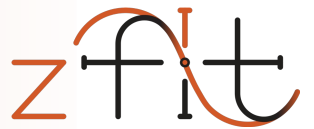
What happens *exactly*?

1) Function is called with a *signature* (Tensors, Python objects)
2) Function is traced:
   - Python code is executed
   - TF code is remembered in graph
   - Caches graph with signature
3) Execute graph with inputs

```
@tf.function(autograph=False)
def add_mult(a, b, c, d):
    print("compiling...")
    tf.print("running")
    sum_ab = a + b
    sum_cd = c + d
    return sum_ab * sum_cd
```

It wasn't always that easy!
(TF1, session, ...)

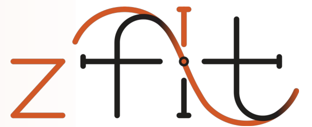
Backend in zfit



- TensorFlow: can wrap arbitrary Python function
- zfit can switch:
  - Graph compiled or eager (Python like)
  - Autograd or numerical gradient

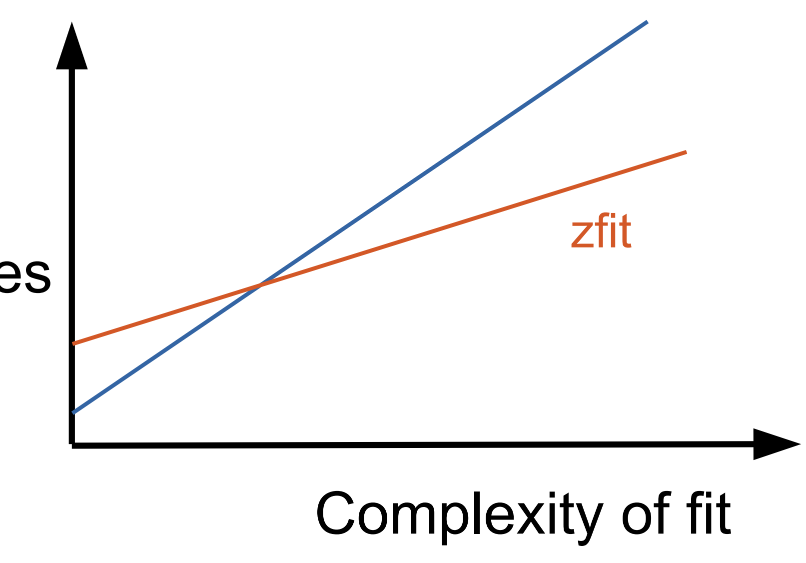
→ arbitrary Python code supported

Scalable: Usability

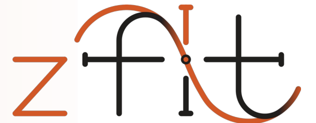


Things should not be easy or hard, but consistent

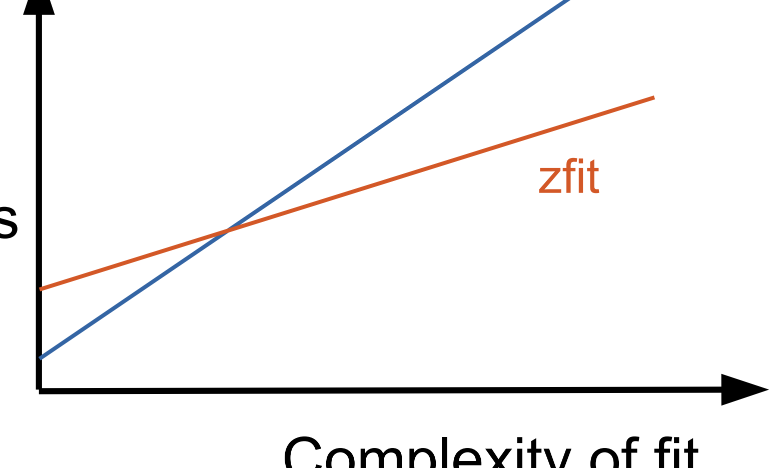
- Code lines
  - 5 or 10: irrelevant
  - 50 or 300: matters



Scalable: Usability

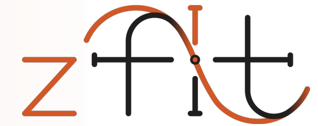


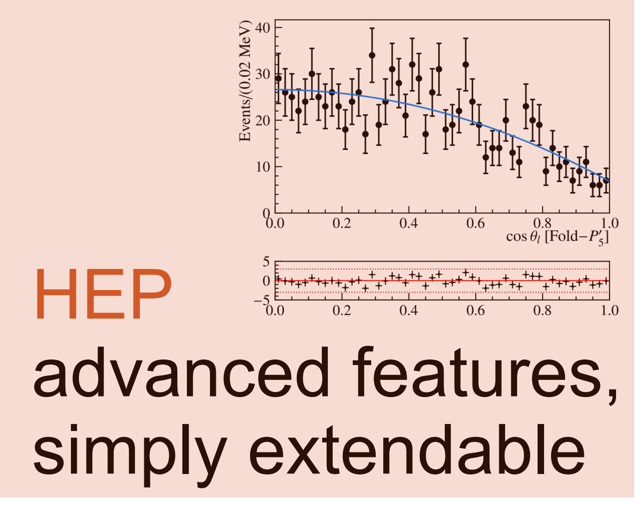
Things should not be easy or hard, but consistent



- Code lines
  - 5 or 10: irrelevant
  - 50 or 300: matters
- Cover all usecases out of the box is impossible
  - Convenient base classes, allow full control
  - **Modular structure**; provide all elements

HEP Model Fitting in Python





HEP
advanced features,
simply extendable

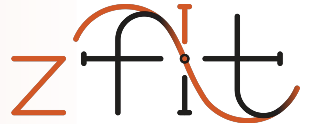
Scalable
large data, complex models

Pythonic

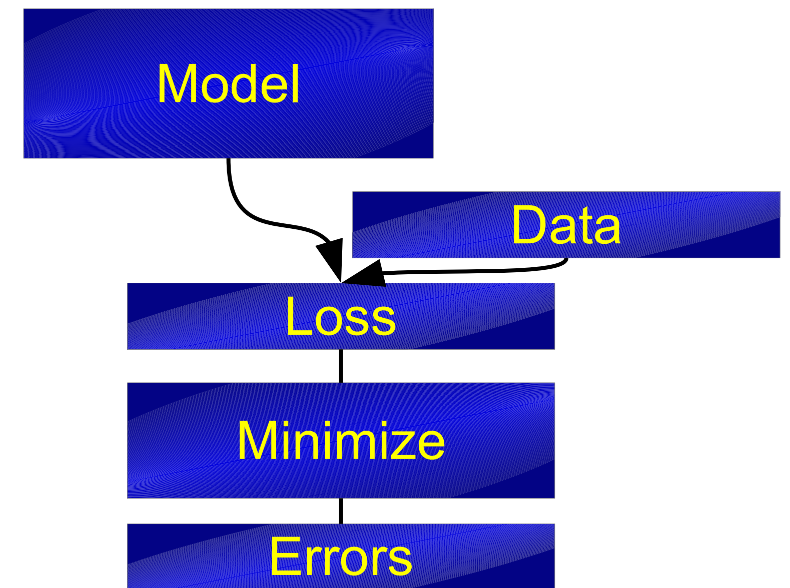
NumPy python™

integrate into ecosystem, stable API

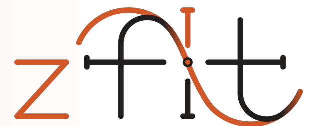
Complete fit



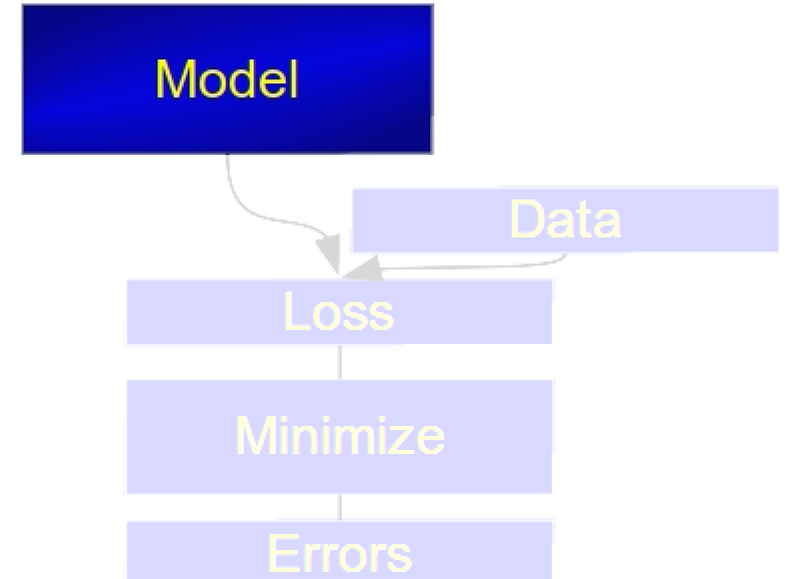

```python
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



Complete fit: Model

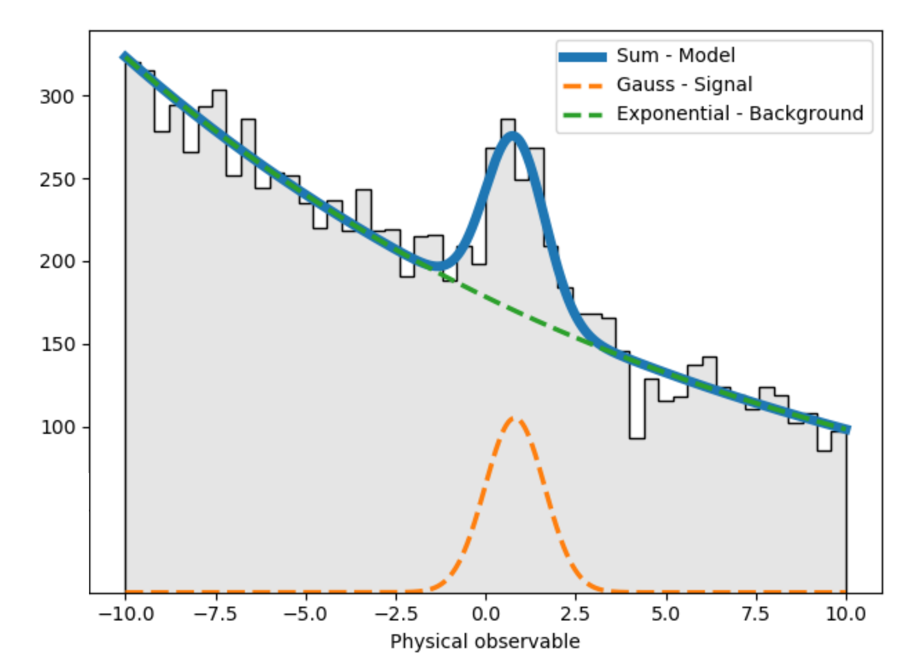


```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



Example: Mass fit

- Sum, Product, *(Convolution)*
- Gauss, (double) Crystalball,...
- Exponential, Polynomials,…
- Histograms, SplineInterpolation,...




```
lambd = zfit.Parameter("lambda", -0.06, -1, -0.01)
frac = zfit.Parameter("fraction", 0.3, 0, 1)

gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
exponential = zfit.pdf.Exponential(lambd, obs=obs)
model = zfit.pdf.SumPDF([gauss, exponential], fracs=frac)
```

Example: Mass fit

- Sum, Product, *(Convolution)*
- Gauss, (double) Crystalball,...
- Exponential, Polynomials,…
- Histograms, SplineInterpolation,...

```
lambd = zfit.Parameter("lambda", -0.06, -1, -0.01)
frac = zfit.Parameter("fraction", 0.3, 0, 1)

gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
exponential = zfit.pdf.Exponential(lambd, obs=obs)
model = zfit.pdf.SumPDF([gauss, exponential], fracs=frac)
```

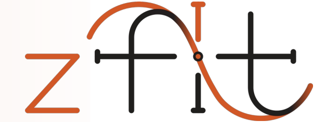
Good for out-of-the-box but…
does not cover even closely all HEP PDFs

Custom PDF

```python
from zfit import z
from zfit.z import numpy as znp

class CustomPDF(zfit.pdf.ZPDF):
    _PARAMS = ['alpha']

    def _unnormalized_pdf(self, x):
        data = z.unstack_x(x)
        alpha = self.params['alpha']

        return znp.exp(alpha * data)
```

implement custom function

Custom PDF



```python
from zfit import z
from zfit.z import numpy as znp

class CustomPDF(zfit.pdf.ZPDF):
    _PARAMS = ['alpha']

    def _unnormalized_pdf(self, x):
        data = z.unstack_x(x)
        alpha = self.params['alpha']

        return znp.exp(alpha * data)
```

```python
custom_pdf = CustomPDF(obs=obs, alpha=0.2)

integral = custom_pdf.integrate(limits=(-1, 2))
sample   = custom_pdf.sample(n=1000)
prob     = custom_pdf.pdf(sample)
```

} use functionality of model

Custom PDF

```python
from zfit import z
from zfit.z import numpy as znp

class CustomPDF(zfit.pdf.ZPDF):
    _PARAMS = ['alpha']

    def _unnormalized_pdf(self, x):
        data = z.unstack_x(x)
        alpha = self.params['alpha']

        return znp.exp(alpha * data)
```

Example of zfit Base Classes

Can also override:

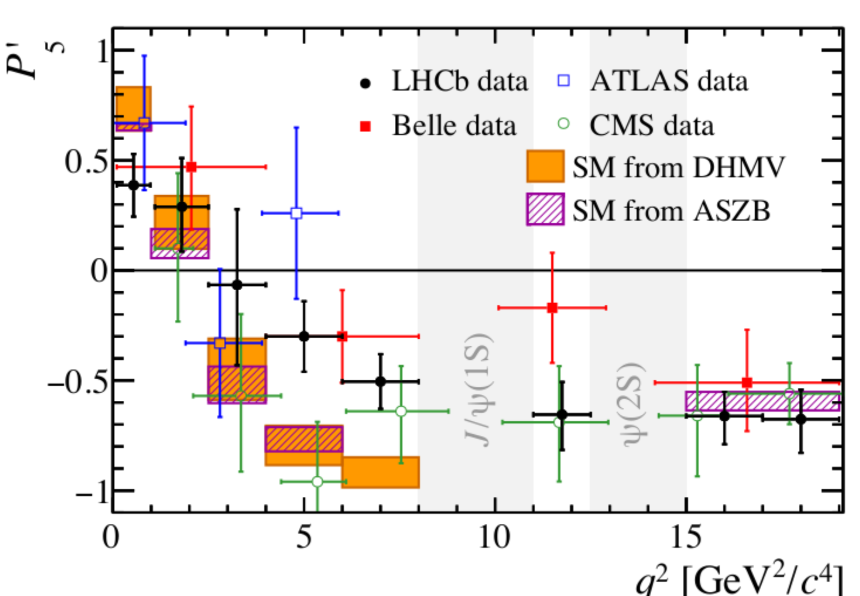
- integrate → _integrate
- pdf → _pdf
- sample → _sample

Or register integral

```python
custom_pdf = CustomPDF(obs=obs, alpha=0.2)

integral = custom_pdf.integrate(limits=(-1, 2))
sample   = custom_pdf.sample(n=1000)
prob     = custom_pdf.pdf(sample)
```

} use functionality of model

$B^0 \to K^{*0} l^+ l^-$ angular: P5'

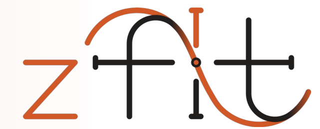



```python
class P5pPDF(zfit.pdf.ZPDF):
    _PARAMS = ['FL', 'AT2', 'P5p']
    _N_OBS = 3

    def _unnormalized_pdf(self, x):
        FL = self.params['FL']
        AT2 = self.params['AT2']
        P5p = self.params['P5p']
        costheta_l, costheta_k, phi = ztf.unstack_x(x)

        sintheta_k = tf.sqrt(1.0 - costheta_k * costheta_k)
        sintheta_l = tf.sqrt(1.0 - costheta_l * costheta_l)

        sintheta_2k = (1.0 - costheta_k * costheta_k)
        sintheta_2l = (1.0 - costheta_l * costheta_l)

        sin2theta_k = (2.0 * sintheta_k * costheta_k)
        cos2theta_l = (2.0 * costheta_l * costheta_l - 1.0)

        pdf = ((3.0 / 4.0) * (1.0 - FL) * sintheta_2k +
               FL * costheta_k * costheta_k +
               (1.0 / 4.0) * (1.0 - FL) * sintheta_2k * cos2theta_l +
               -1.0 * FL * costheta_k * costheta_k * cos2theta_l +
               (1.0 / 2.0) * (1.0 - FL) * AT2 * sintheta_2k *
               sintheta_2l * tf.cos(2.0 * phi) + tf.sqrt(FL * (1 - FL))
               * P5p * sin2theta_k * sintheta_l * tf.cos(phi))

        return pdf
```

P5': optimised observable
Fit of P5', from [1, 2]

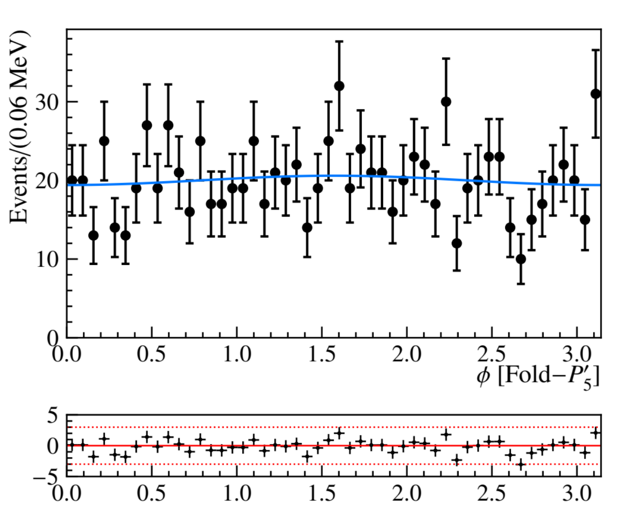
[1] JHEP 02 (2016) 104 [2] Phys.Rev.Lett. 118 (2017) no.11, 111801

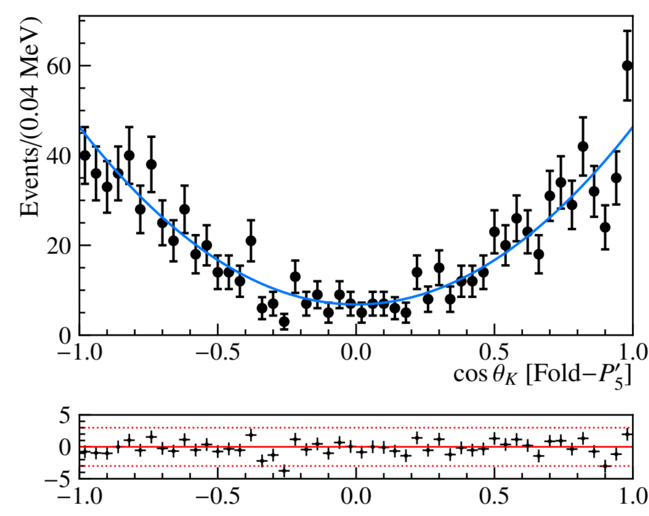
$B^0 \to K^{*0} l^+ l^-$ angular: fitted P5'

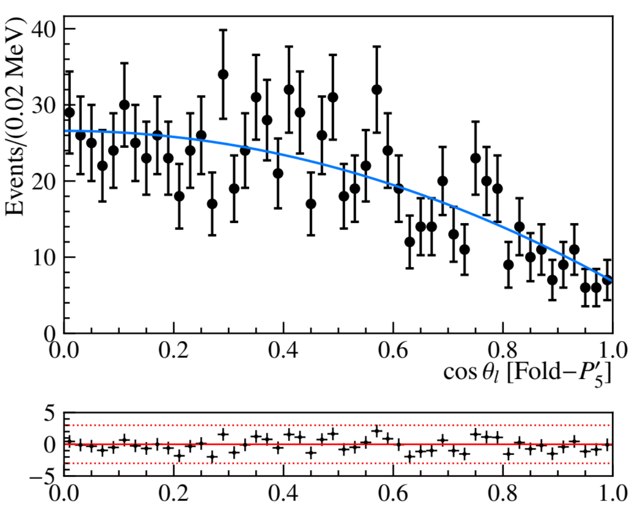


Projections of three angles

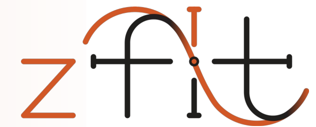
Plot with mplhep, matplotlib







Binned models



- Closely modelled and compatible with boost-histogram/hist
  - Axes, names, ….
- Have "counts" and "rel_counts" method (returns hist-like)

```
h = hist.Hist(hist.axis.Regular(3, -3, 3, name="x", flow=False),
              hist.axis.Regular(2, -5, 5, name="y", flow=False))
x = np.random.randn(1_000_000)
y = 0.5 * np.random.randn(1_000_000)
h.fill(x=x, y=y)

pdf = zfit.pdf.HistogramPDF(data=h)
```

...and back

```
h_back = pdf.to_hist()
```

```
mplhep.histplot(h_back)
```

Binned models

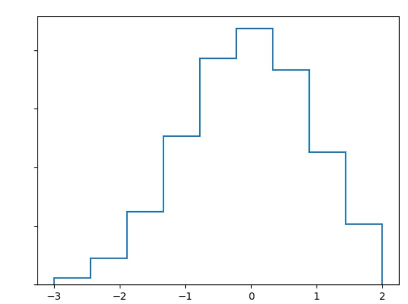
- Closely modelled and compatible with boost-histogram/hist
  - Axes, names, ….
- Have "counts" and "rel_counts" method (returns hist-like)

```
h = hist.Hist(hist.axis.Regular(3, -3, 3, name="x", flow=False),
              hist.axis.Regular(2, -5, 5, name="y", flow=False))
x = np.random.randn(1_000_000)
y = 0.5 * np.random.randn(1_000_000)
h.fill(x=x, y=y)

pdf = zfit.pdf.HistogramPDF(data=h)
```

...and back

```
h_back = pdf.to_hist()
```

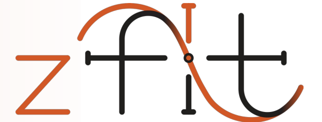
```
mplhep.histplot(h_back)
```

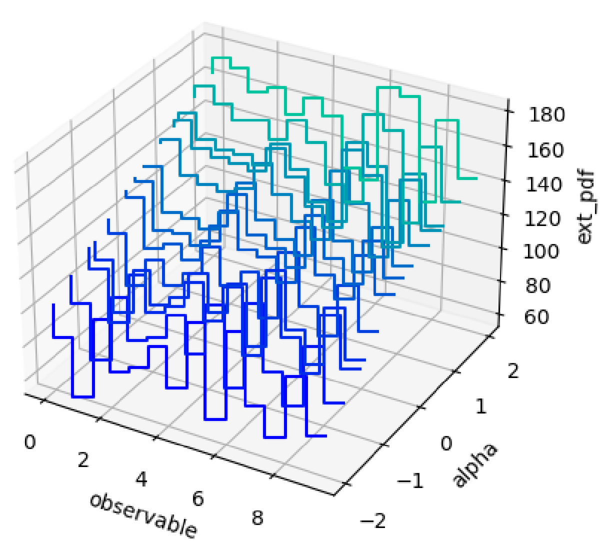


Change the yield

```
ntot = zfit.Parameter("ntot", 1_000)
pdf = zfit.pdf.HistogramPDF(h, extended=ntot)
```

More histograms

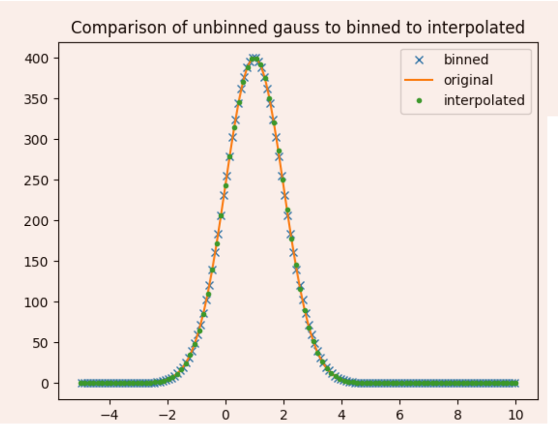




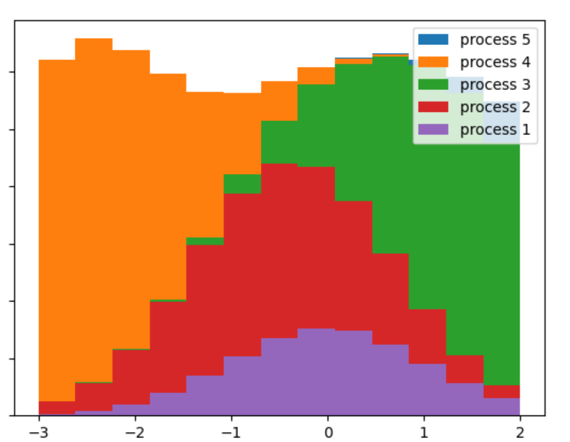

```
pdfs = [zfit.pdf.HistogramPDF(h) for h in histos]
alpha = zfit.Parameter('alpha', 0, -5, 5)
morph = SplineMorphingPDF(alpha=alpha, hists=pdfs)
```

Shape modifier

```
pdf_syst = zfit.pdf.BinwiseScaleModifier(pdf, modifiers=True)
```

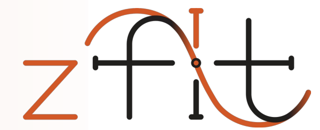


Unbinned → binned → interpolated

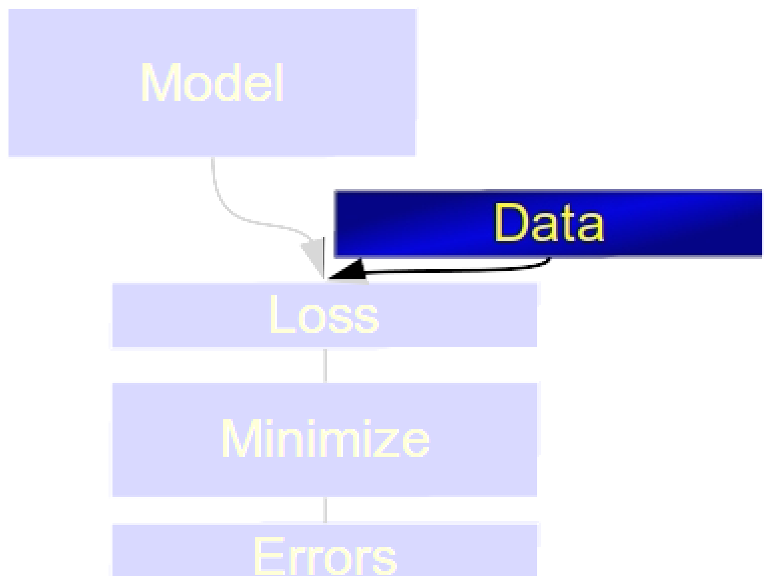



```
pdfs = [zfit.pdf.HistogramPDF(h) for h in histos]
sumpdf = zfit.pdf.BinnedSumPDF(pdfs)
```

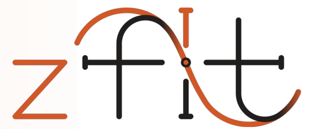
Complete fit: Data




```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



Complete fit: Data



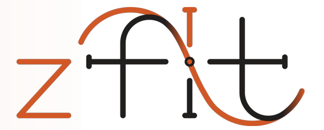
- From different sources
  - Hist, numpy, Pandas, ROOT, …

Use the HEP/Python ecosystem for preprocessing

- Sampled from a model (toy studies)

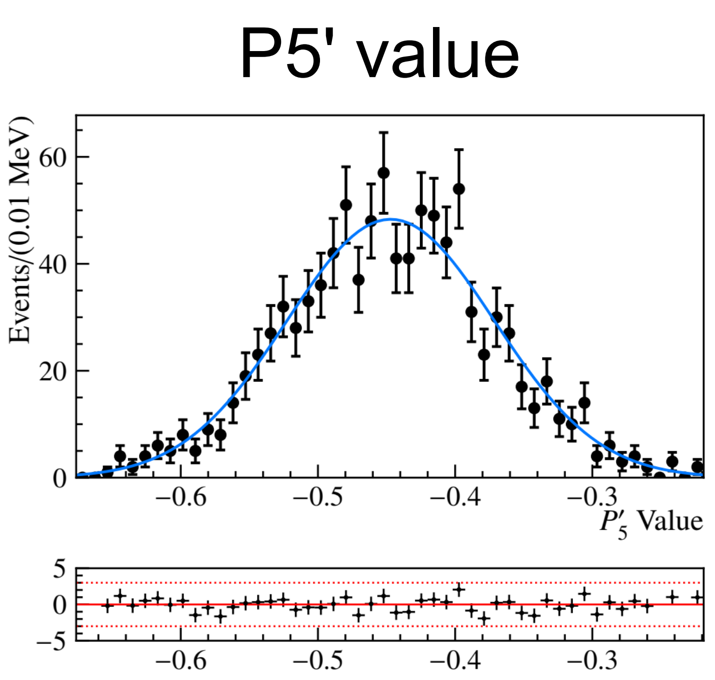
```
data = model.create_sampler(n_sample, limits=obs)
```

$B^0 \to K^{*0} l^+ l^-$ angular: toy study

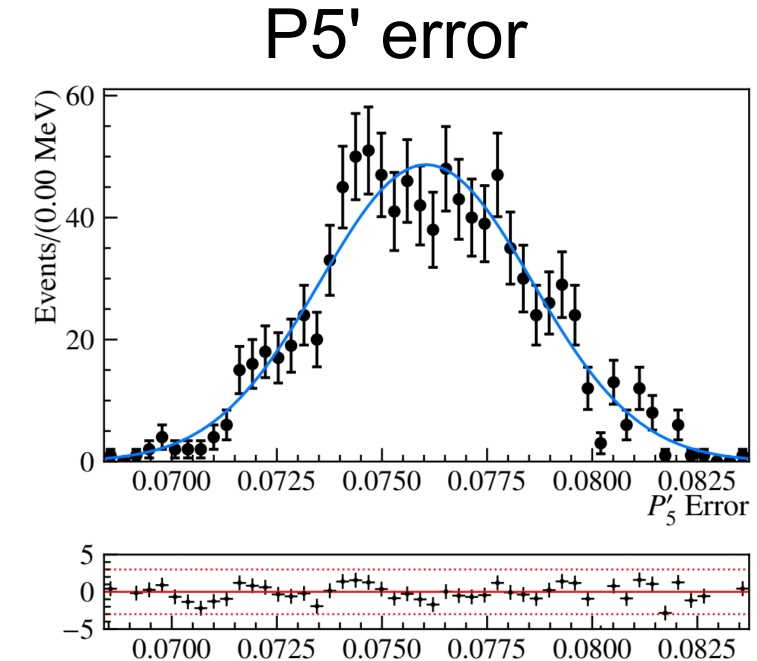


Result of toy study

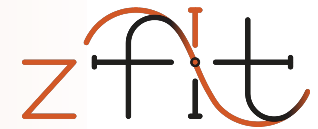
P5' value



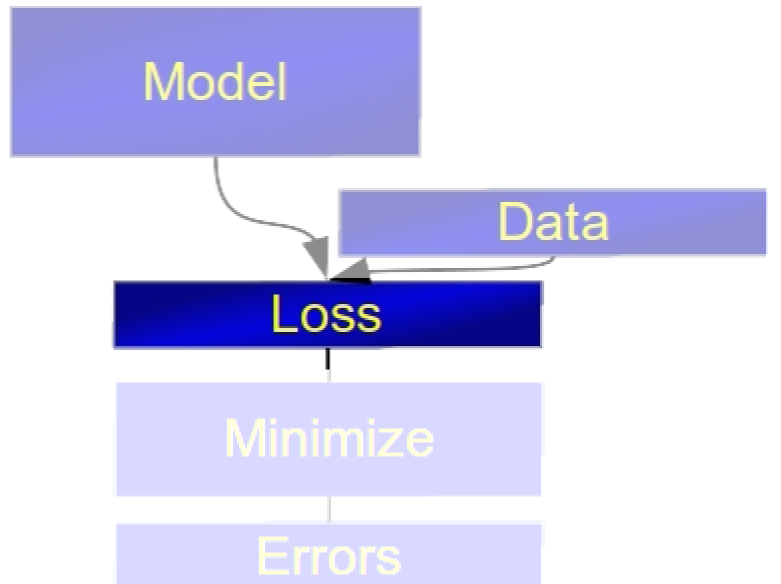
P5' error



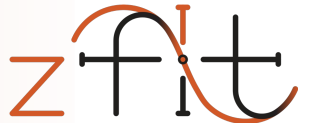
Complete fit: Loss




```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



Loss



```
mu_shared = zfit.Parameter("mu_shared", 1., -4, 6)
sigma1 = zfit.Parameter("sigma_one", 1., 0.1, 10)
sigma2 = zfit.Parameter("sigma_two", 1., 0.1, 10)

gauss1 = zfit.pdf.Gauss(mu=mu_shared, sigma=sigma1, obs=obs)
gauss2 = zfit.pdf.Gauss(mu=mu_shared, sigma=sigma2, obs=obs)
```

shared parameters

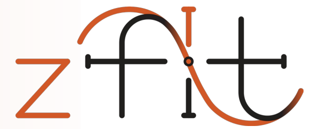
```
nll_simultaneous = zfit.loss.UnbinnedNLL(model=[gauss1, gauss2],
                                         data=[data1, data2])

nll1 = zfit.loss.UnbinnedNLL(model=gauss1, data=data1)
nll2 = zfit.loss.UnbinnedNLL(model=gauss2, data=data2)
nll_simultaneous2 = nll1 + nll2
```

Completely equivalent

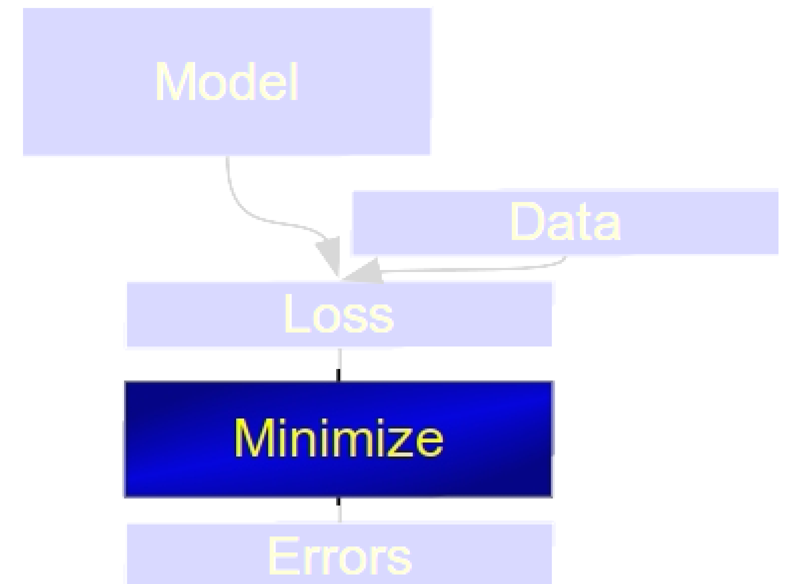
(arbitrary) constraints supported, manually added to loss

```
constr = GaussianConstraint(params=params, observation=observed, uncertainty=sigma)
nll = zfit.loss.BinnedNLL(model=model, data=data, constraint=constr)
```

Complete fit: Minimization



```
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```



Minimize

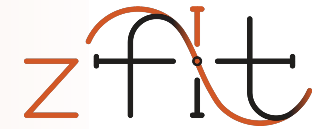
- Problem: many, non-unified minimizer APIs
  - SciPy inferface "a bit messy", different convergence criterion, etc...
- Unified API: zfit minimizers, simply switch

```
minimizer = zfit.minimize.IpyoptV1()
minimizer = zfit.minimize.Minuit()
minimizer = zfit.minimize.ScipyTrustConstrV1()
minimizer = zfit.minimize.NLoptLBFGSV1()
```

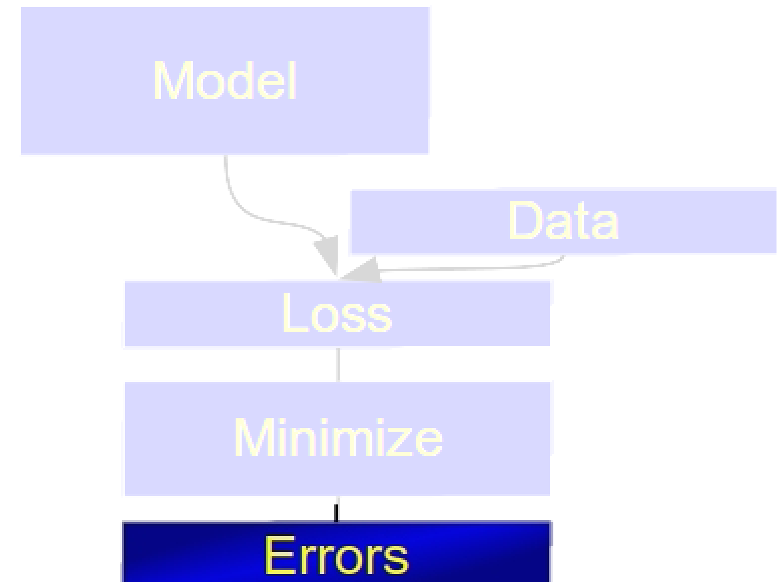
- Can use zfit loss, but also ***pure Python function***

```
result = minimizer.minimize(func, params)
```

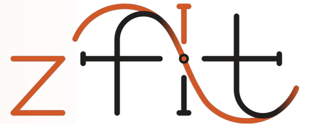
Complete fit: Result



```python
normal_np = np.random.normal(2., 3., size=10_000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.hesse()
param_errors_asymmetric, new_result = result.errors()
```

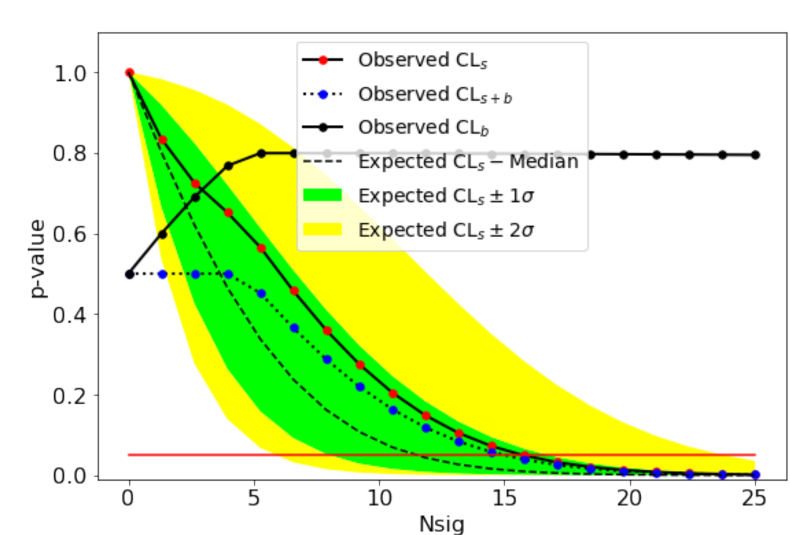


Back to HEP ecosystem: hepstats

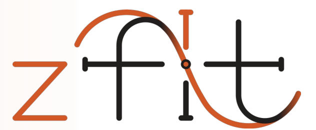


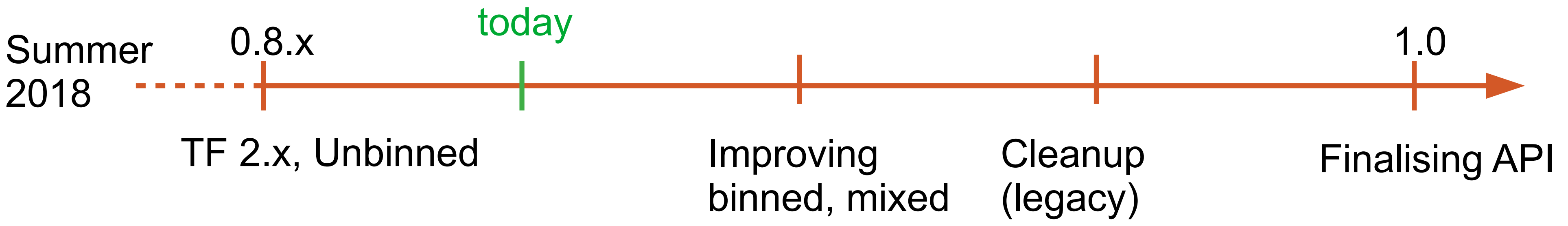
- Inference library for hypothesis tests
- Takes model, data, loss from zfit
- sWeights, CI, limits, …
  asymptotic or toys

```
calculator = AsymptoticCalculator(loss, minimizer)
poinull = POIarray(Nsig, np.linspace(0.0, 25, 20))
poialt = POI(Nsig, 0)
ul = UpperLimit(calculator, poinull, poialt)
ul.upperlimit(alpha=0.05, CLs=True)
```



zfit: status



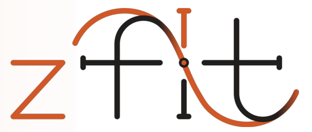


A lot of experience and proven API, but also technical debts (global parameters, ...)

Quite performant, but specialized libraries better (pyhf ~100x faster for simple example)

Can perform a diverse set of fits (LHCb, Belle 2, …, Amplitudes, time-dependend, templated,…) in Python in reasonable time

Conclusion



build stable model fitting ecosystem for HEP

- Integrate into HEP ecosystem

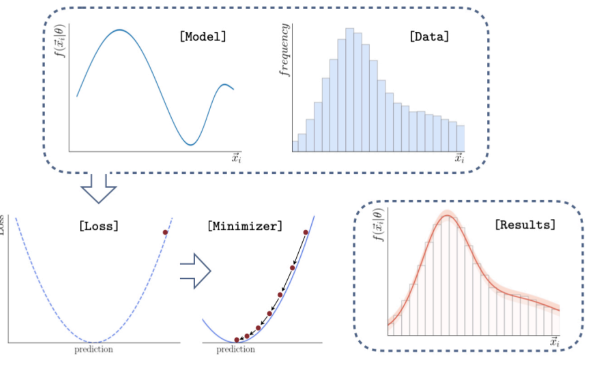
  functionality limited; stable API

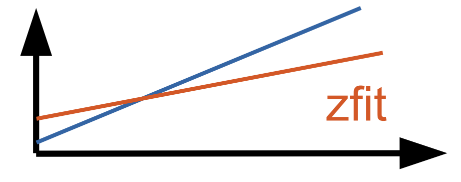
- Technical requirements

  performance; maintainability

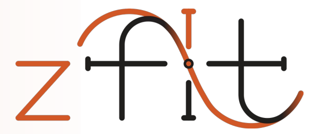
- HEP requirements

  advanced features; simply extendable code





Sources



- LHCb collision: https://physicsworld.com/wp-content/uploads/2018/08/LHCb-collision.png

Backup Slides

https://zfit.github.io/zfit/

zfit@GitHub

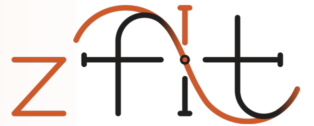
Gitter channel

zfit@physik.uzh.ch

Join the discussion!

- Backend & TF
- Amplitude
- K*ll toys
- K*mumu Wilson coeffs
- Other fitting packages
- Zfit (associated) packages
- Zfit project
- Zfit elements examples

Fitting in Python



A lot of projects are around

TensorProb

TensorFlow Analysis

probfit

RooFit

carl

mlfit

scipy

TensorFlow Probability

HEP Python

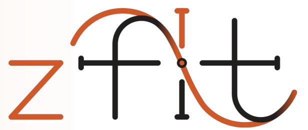
Non-HEP

Backend & TensorFlow

Backend: a comparison

- TensorFlow: supports the most features to this day
- PyTorch: missing advanced math (complex support, …)
- Numpy/SciPy: Too slow, no gradient, no GPU
- JAX: very promising, but *no globals (cache,…)*,
  only static known shapes (adaptive algorithms, accept-reject...), only JAX/Numpy arrays compatible
- SymPy: limited to mathematical expressions (no control-flow,…)
  but can convert to any other backend (used by TensorWaves)

Backend: tracing and autograd



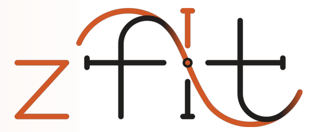
Tracing
execute Python once, remember (algebraic) computation

Autograd
"analytic" gradient of function

Recent rise of big data industry created libraries that support this

Includes GPU support, optimizations, caching,...

zfit: the project



build *the* stable model fitting ecosystem for HEP

- Integrate into HEP ecosystem

  functionality limited; stable API

- Technical requirements

  performance; maintainability

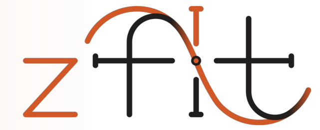
- HEP requirements

  advanced features; simply extendable code

Deep Learning

lessons for model fitting

Deep Learning



Neural Network

„One huge, complicated function“

[Model]

[Data]

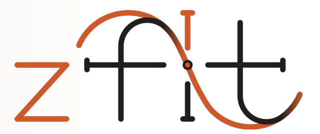
„Big Data“

[Loss]

[Minimizer]

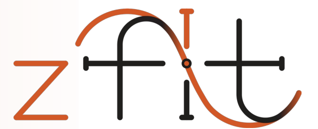
[Results]

Main backend: TensorFlow



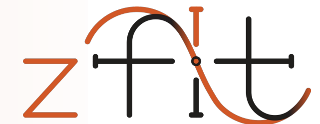
- By Google, highly popular (150k★, 4th on GitHub)

Main backend: TensorFlow



- By Google, highly popular (130k★, 4th on [GitHub])
- Used in multiple physics libraries and analyses

Main backend: TensorFlow

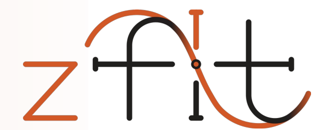


- By Google, highly popular (150k★, 4th on [GitHub icon])
- Consists of "two parts":
  - High level API for building neural networks *(NOT used!)*
  - Low level API with Numpy-style syntax

    tf.sqrt, tf.random.uniform,…
- Two modes:
  - "numpy"-like (full Python flexibility)
  - "compiled" (very performant)

} GPU/Multi CPU support

Delegating the workload



	C++ library (RooFit,...)	Numpy based	zfit
HEP specific content/API			
Models		SciPy	TF Probability
Gradients			TensorFlow
Computational optimizations			
Parallelization/GPU		Numba NumPy	intel NVIDIA
Low level handling		python	

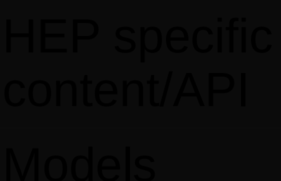
Delegating the workload

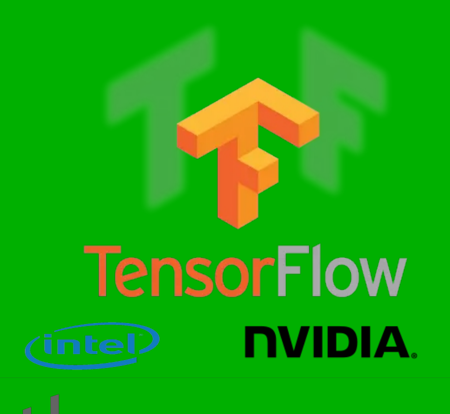
	C++ library (RooFit,...)	Numpy based	zfit
HEP specific content/API			
Models		SciPy	TF Probability
Gradients			TensorFlow
Computational optimizations			
Parallelization/GPU		Numba NumPy	intel NVIDIA
Low level handling		python	

"Stepping on the shoulders of a giant"

Delegating the workload

	C++ library (RooFit,...)	Numpy based	zfit
HEP specific content/API			API & Workflow
Models		SciPy	TF Probability
Gradients			TensorFlow
Computational optimizations			
Parallelization/GPU		Numba NumPy	intel NVIDIA
Low level handling		python	python



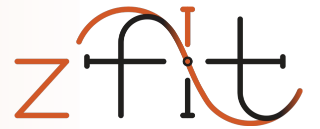


Used & maintained (!) by industry

Can we express model fitting as static graphs?

Yes!

HPC perspective



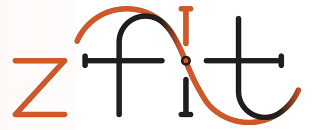
1) Definition of computation, shape etc. (add static knowledge)
2) Compilation of the graph
3) Execution of computation (re-use optimized graph)

Inside TF, hidden to end-user

HPC: the more is know *before* the execution, the better

TensorFlow takes care of *how* to use this knowledge

Graph elements



… do not have to be constant!

Parameters

Can change their value

Random numbers

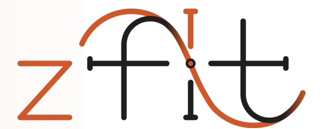
Generate newly on every graph execution: MC integration,…

Control flow (if, while)

Steer the execution: Accept-reject sampling (while), etc.

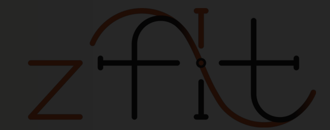
Static, not constant

Deep Learning vs. Model Fitting



Similarity	Complicated Models	Large Data	Composed loss	Minimization	Results and uncertainties
HEP	Non-trivial functions	Whole Dataset	simultaneous, constraints	Global min, 2nd derivative algorithm	Hesse, profiling
Deep Learning	Combine many, trivial functions	Many, small Batches	*Anything! (GANs, RL,...)*	Local (!) min, 1th derivative, many steps	None
Conclusion					

Deep Learning vs. Model Fitting



Similarity`	Complicated Models	Large Data	Composed loss	Minimization	Results and uncertainties
HEP	Non-trivial functions	Whole Dataset	simultaneous, constraints	Global min, 2nd derivative algorithm	Hesse, profiling
Deep Learning	Combine many, trivial functions	Many, small Batches	*Anything! (GANs, RL,...)*	Local (!) min, 1th derivative, many steps	None
Conclusion	No real impact	Optimizations for OOM calculations	HEP trivial special case	~~Optimizers~~ Free „analytic“ derivatives!	No support, but simple

But...

what ***is*** a Deep Learning library?

Deep Learning vs. Model Fitting

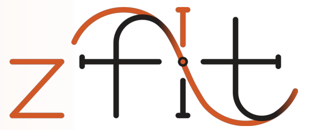
Similarity	Complicated Models	Large Data	Composed loss	Minimization	Results and uncertainties
HEP	Non-trivial functions	Whole Dataset	simultaneous, constraints	Global min, 2nd derivative algorithm	Hesse, profiling
Deep Learning	Combine many, trivial functions	Many, small Batches	*Anything! (GANs, RL,...)*	Local (!) min, 1th derivative, many steps	None
Conclusion	No real impact	Optimizations for OOM calculations	HEP trivial special case	~~Optimizers~~ „analytic“ derivatives!	No support, but simple

Deep Learning vs. Model Fitting

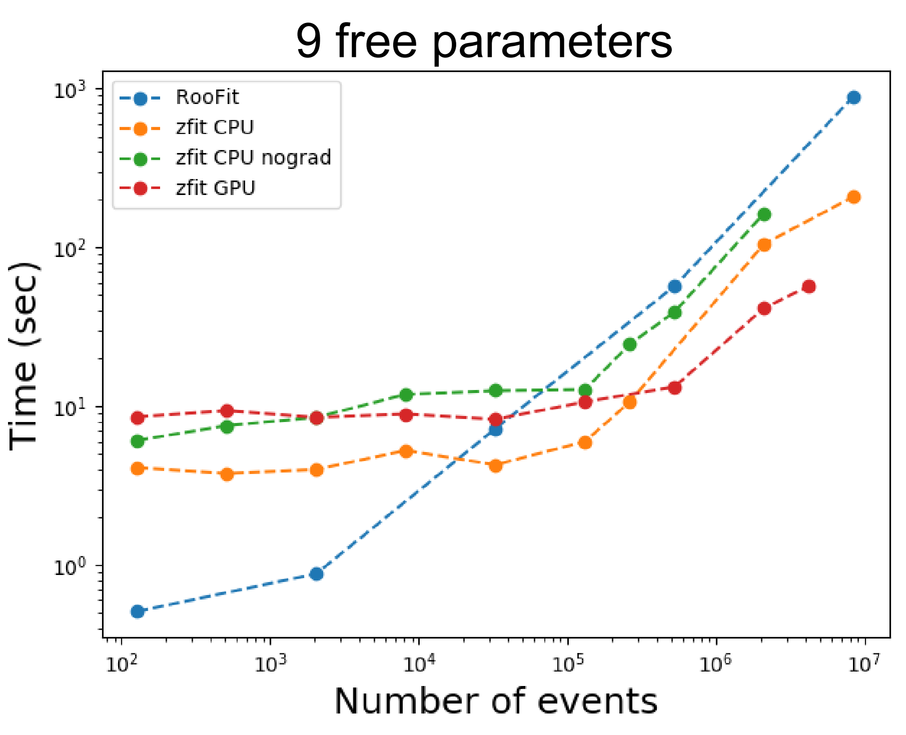
Similarity	Complicated Models	Large Data	Composed loss	Minimization	Results and uncertainties
HEP	Non-trivial functions	Whole Dataset	simultaneous, constraints	Global min, 2nd derivative algorithm	Hesse, profiling
Deep Learning	Combine many, trivial functions	Many, small Batches	*Anything! (GANs, RL,...)*	Local (!) min, 1th derivative, many steps	None
Conclusion	No real impact	Optimizations for OOM calculations	HEP trivial special case	~~Optimizers~~ „analytic“ derivatives!	No support, but simple

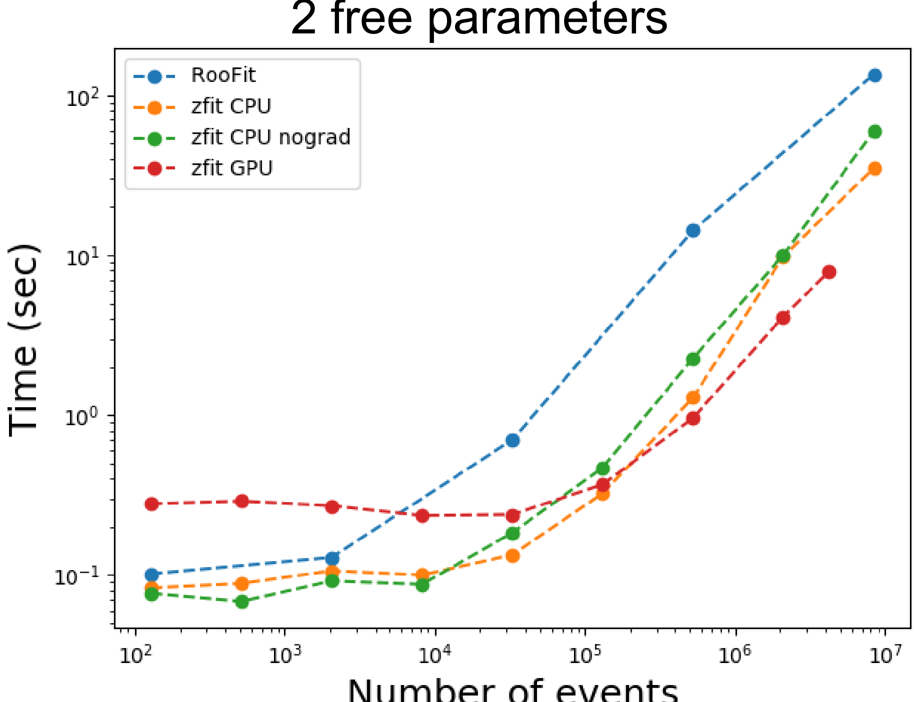
Modern, high performance computing

Scalability: Performance



Fitting time (lower is better): **RooFit** vs. **zfit**

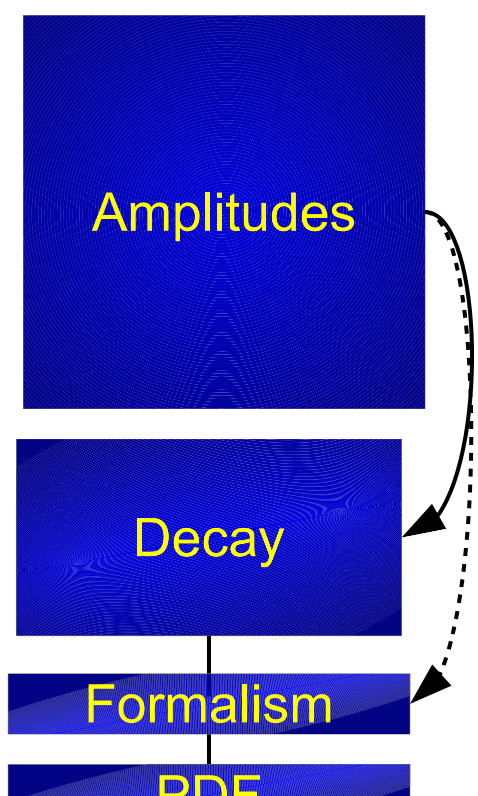


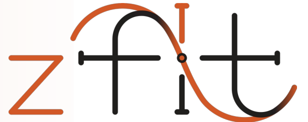


Amplitude

Example amplitude

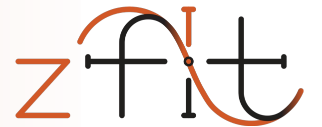
```python
RESONANCES = [('rho(770)', ('pi-', 'pi0'), bw_amplitude),
              ('K(2)*(1430)0', ('K+', 'pi-'), bw_amplitude),
              ('K(0)*(1430)+', ('K+', 'pi0'), bw_amplitude),
              ('K*(892)+', ('K+', 'pi0'), bw_amplitude),
              ('K(0)*(1430)0', ('K+', 'pi-'), bw_amplitude),
              ('K*(892)0', ('K+', 'pi-'), bw_amplitude)]

COEFFS = {...}

D2Kpipi0 = Decay('D0', ['K+', 'pi-', 'pi0'])

for res, children, amp in RESONANCES:
    D2Kpipi0.add_amplitude(res, children, amp, COEFFS[res])

formalism = ThreeBodyDalitzFormalism("Zemach B Frame")

pdf = D2Kpipi0.create_pdf(name="D2Kpipi0", formalism=formalism)
```





Angular toys

$B^0 \to K^{*0} l^+ l^-$ angular: toy study

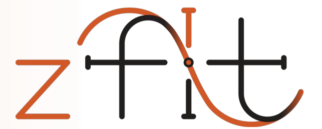


Sensitivity study

- draw toys (sample) from PDF
- Fit to sample

```
for i in range(ntoys):

    # set initial sampling values
    for param in params:
        param.set_value(...)

    sampler.resample()

    # set random initial values
    for param in params:
        param.set_value(...)

    result = minimizer.minimize(nll)

    if result.converged:
        ...
```

$B^0 \to K^{*0} l^+ l^-$ angular: toys

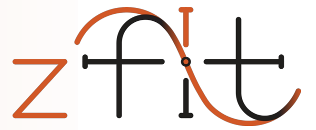


Sensitivity study

- draw toys (sample) from PDF
- Fit to sample

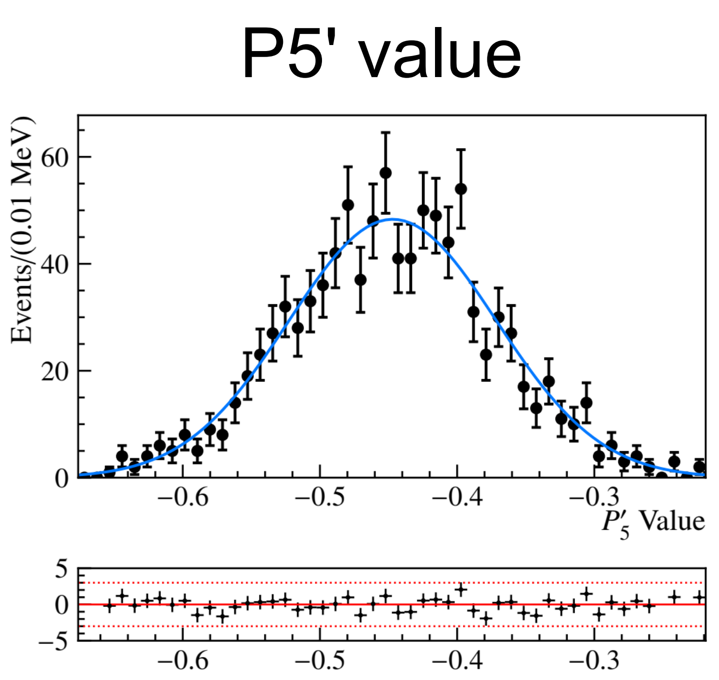
```
for i in range(ntoys):

    # set initial sampling values
    for param in params:
        param.set_value(...)

    sampler.resample()

    # set random initial values
    for param in params:
        param.set_value(...)

    result = minimizer.minimize(nll)

    if result.converged:
        ...
```

$B^0 \to K^{*0} l^+ l^-$ angular: toy study

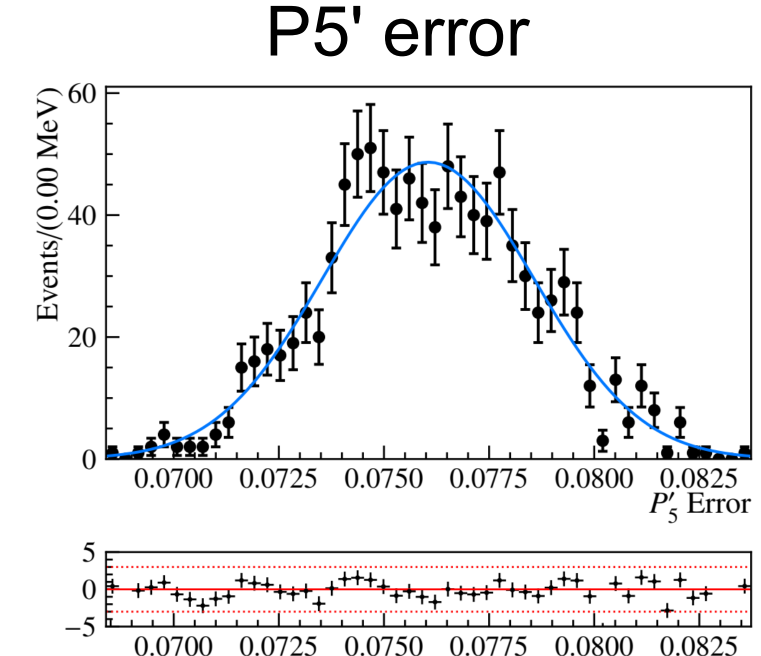


Result of toy study

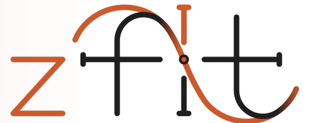
P5' value



P5' error



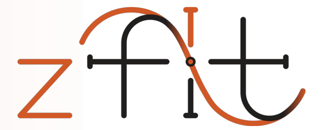
Extending with a mass shape



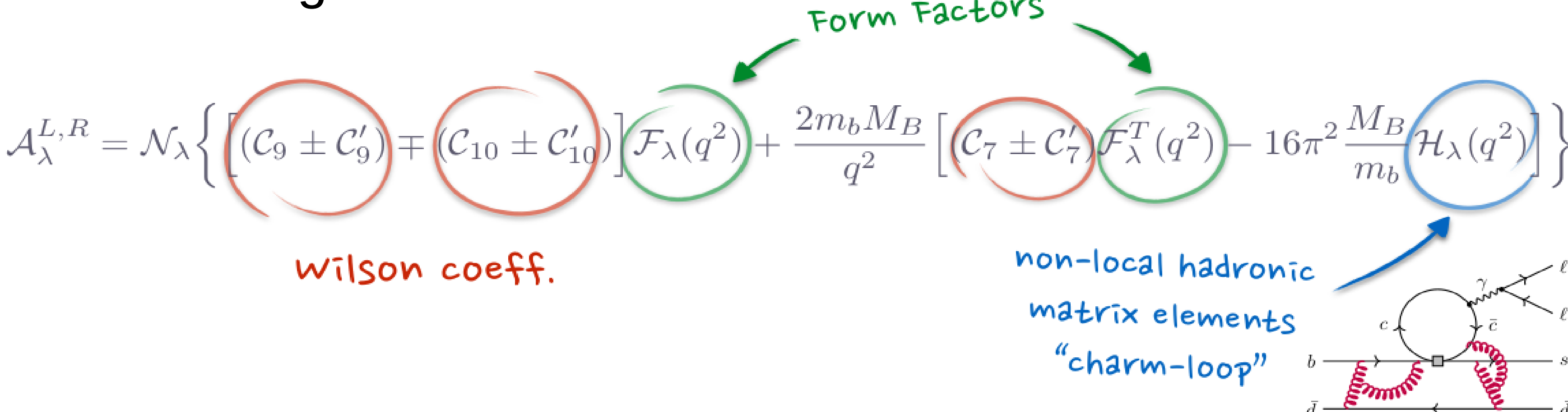
```
# Create mass pdf
mu = zfit.Parameter("mu", 5279, 5200, 5400)
sigma = zfit.Parameter("sigma", 30, 0, 300)
a0 = zfit.Parameter("a0", 1.0, 0, 10)
a1 = zfit.Parameter("a1", 1.0, 0, 10)
n0 = zfit.Parameter("n0", 5, 0, 10)
n1 = zfit.Parameter("n1", 5, 0, 10)

mass = zfit.Space("mass", limits=(4900, 5600))

massPDF = zfit.pdf.DoubleCB(obs=mass, mu=mu, sigma=sigma,
                            alphal=a0, nl=n0, alphar=a1, nr=n1)

pdf = massPDF * angularPDF
```

Build model

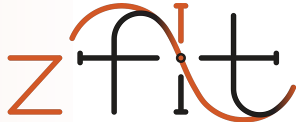
$B^0 \to K^{*0}\mu^+\mu^-$ full amplitude



- Measuring the full differential decay ratio [1, 2]
  - Angular, q^2 distribution $\frac{d^4\Gamma}{dq^2\, d\cos\theta_\ell\, d\cos\theta_K\, d\phi} \propto \sum J_i(q^2) f(\cos\theta_\ell, \cos\theta_K, \phi)$
  - Branching ratio information

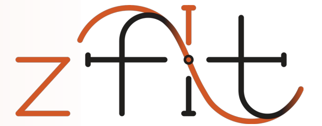


[1] arXiv:1805.06378 [2] arXiv:1805.06401



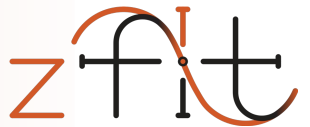
Fitting libraries and comparison

Python model fitting in HEP



- Scalable: large data, complex models
- Pythonic: use Python ecosystem/language
- Specific HEP functionality:
  - Normalization: specific range, numerical integration,...
  - Composition of models
  - Multiple dimensions
  - Custom models
  - Non-trivial loss (constraints, simultaneous,…)

RooFit



- *Limited customization and extendibility*
- *Sub-optimal scalability for ever larger datasets and modern computing infrastructure*
- Isolated, aging ecosystem, no cutting-edge software
- Not Python native
  - *Memory allocation errors*
  - *Arbitrary C++ limitations*
  - *No real integration into the Python ecosystem*

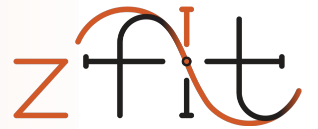
HEP Python projects

Probfit, TensorProb,…

- Lack generality and extendibility
- “experimental”, but great proof of concept
  - API and Python in general
  - Computational backends (e.g. Cython, TensorFlow)
  - Building an ecosystem (iminuit,…)

General impression in comparison with other HEP packages

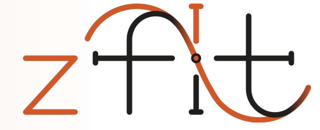
Non-HEP



Scipy, lmfit, TensorFlow Probability,...

- Lack of specific HEP features
  - *Normalization: specific range, numerical integration,...*
  - *Composition of models*
  - *Multiple dimensions*
  - *Custom models*
- Irrelevant functionality supported in API
  - Survival function, …

TFA: approach & differences



- Build «optimized» TensorFlow
  - accept-reject as tf.while_loop, Dataset input,...
- ...and hide the tedious, unambiguous parts
  - automatic normalization, Tensor cache, ...
- Well defined structures, e.g.
  - String name order (like columns) in PDFs, data, limits,...
    - pdf(„x“) * pdf(„y“) => pdf(„x“, „y“)
1-dim 1-dim 2-dim
  - Local/recursive dependency resolution of Parameters

Zfit related packages

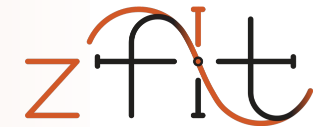
phasespace

- Package for phasespace generation of particles
- Covers functionality of TGenPhaseSpace (and more)
- Pure Python (& TensorFlow), integrates seemless with zfit

```
pion = GenParticle('pi+', PION_MASS)
kaon = GenParticle('K+', KAON_MASS)
kstar = GenParticle('K*', KSTARZ_MASS).set_children(pion, kaon)
gamma = GenParticle('gamma', 0)
bz = GenParticle('B0', B0_MASS).set_children(kstar, gamma)

weights, particles = bz.generate(n_events=1000)
```

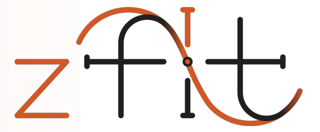
Zfit: project description

zfit project



- zfit: stable core
  - Unbinned fits, binned WIP
  - n-dim models with integral, pdf, sample
- zfit-physics: HEP specific content
  - BreitWigner, DoubleCB,…
  - Faster development, more content
  - Ideal for contributions
    - Auto testing of new pdfs/func
    - Contribution guidelines

zfit: the project



build stable model fitting ecosystem for HEP

- Integrate into HEP ecosystem

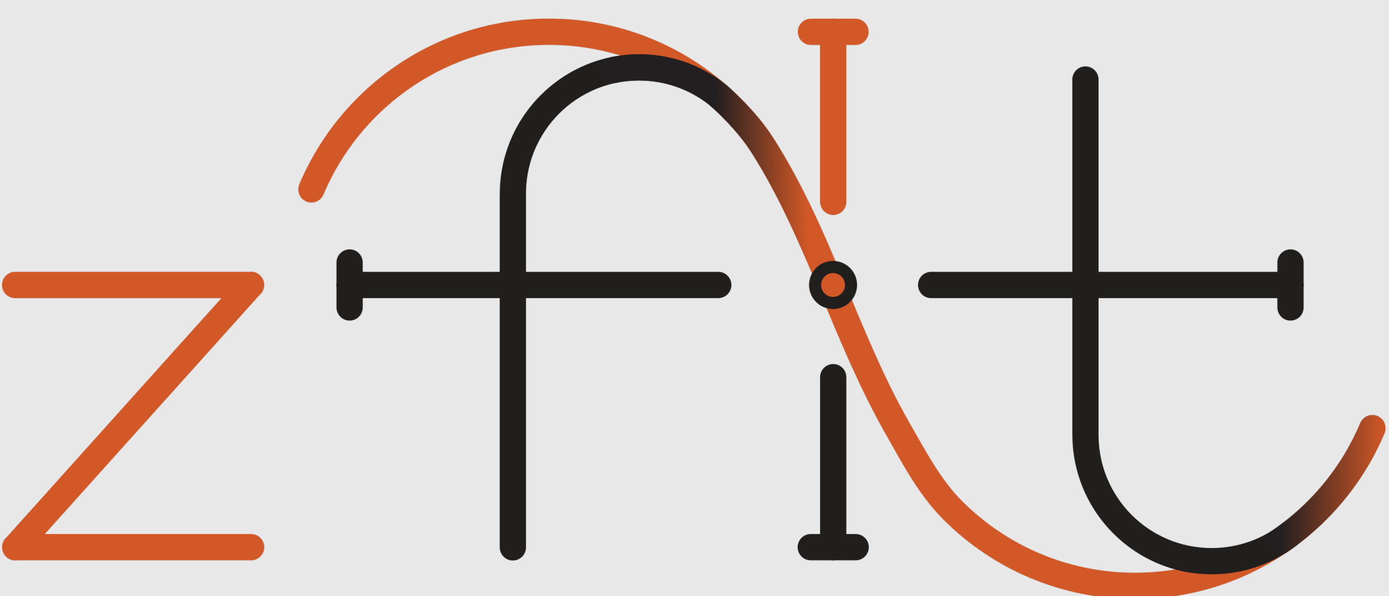
  functionality limited; stable API

- Technical requirements

  performance; maintainability

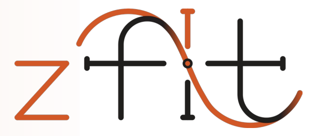
- Analysis requirements

  advanced features; simply extendable code



scalable pythonic fitting

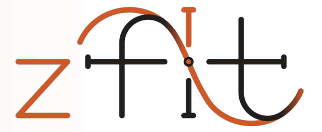
zfit: the project



build *the* stable model fitting ecosystem for HEP

...the time has come

zfit: the project



build *the* stable model fitting ecosystem for HEP

- Integrate into HEP ecosystem

  functionality limited; stable API

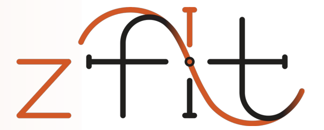
- Technical requirements

  performance; maintainability

- Analysis requirements

  advanced features; simply extendable code

Ecosystem: API & Workflow

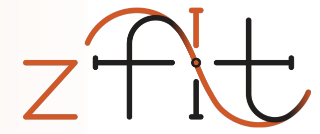


Establish a stable API

- High level libraries (statistics, plotting,...)
  - „code against an interface, not an implementation“
- Replace each component
  - Allow other libraries to implement custom parts

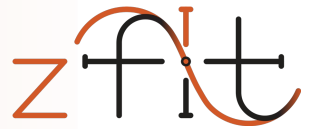
Many discussions with community
to avoid splitting/duplication

Pythonic



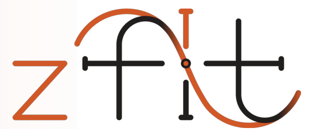
- Pure Python («pip install zfit»)
- Integrated into python ecosystem
    - Load ROOT files (uproot,no ROOT dependence!)
    - Use Minuit for minimization (iminuit)
    - Data preprocessing with Pandas DataFrame
    - Plotting with matplotlib
    - High level statistics (lauztat, more WIP)
- Extendable classes
    - e.g. custom PDF

Scalable



- TensorFlow **hidden** backend, uses graphs
  - numpy-like syntax
  -
    parallelization on CPU/GPU, analytic gradient,...
- Writing functions simple for users *and* developers
  - No Cython, MPI, CUDA,... for *state-of-the-art performance*
  - No low-level maintenance required!
- Used in multiple physics libraries and analyses

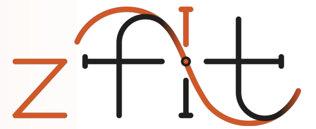
Scalable: TensorFlow



- Deep Learning framework by Google
- Modern, declarative graph approach
- Built for highly parallelized, fast communicating CPU, GPU, TPU,… clusters
- Built to use «Big Data»

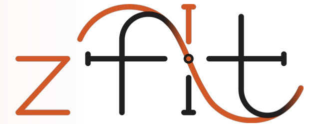
Zfit library examples

Minimize Python function




```python
def func(x):
    x = np.array(x)  # make sure it's an array
    return np.sum((x - 0.1) ** 2 + x[1] ** 4)

func.errordef = 0.5

params = [1, -3, 2, 1.4, 11]

result = minimizer.minimize(func, params)
```

Model, loss building



sum of two pdfs

```
sum_pdf = zfit.pdf.SumPDF([gauss, exponential], fracs=frac)
```

shared parameters

```
mu_shared = zfit.Parameter("mu_shared", 1., -4, 6)

gauss1 = zfit.pdf.Gauss(mu=mu_shared, sigma=sigma1, obs=obs)
gauss2 = zfit.pdf.Gauss(mu=mu_shared, sigma=sigma2, obs=obs)
```

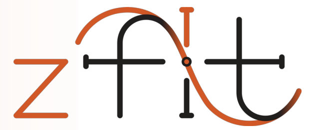
simultaneous loss

```
nll1 = zfit.loss.UnbinnedNLL(model=gauss1, data=data1)
nll2 = zfit.loss.UnbinnedNLL(model=gauss2, data=data2)
nll_simultaneous2 = nll1 + nll2
```

From classical

to more TensorFlow

Model, loss building



Simple combinations

```
func_n = zfit.func.ZFunc(...)  # pseudo code
func = func_1 + func_2 * func_3
```

Composite Parameter

```
pdf = zfit.pdf.Gauss(mu=tensor1, sigma=4)
```

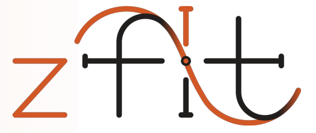
Custom Loss

```
loss = zfit.loss.SimpleLoss(lambda: tensor_loss)
```

up to pure TensorFlow

=> use all of zfit functionality like minimizers

Model building



```
obs = zfit.Space("x", limits=(-10, 10))

mu =    zfit.Parameter("mu",          1,  -4,  6)
sigma = zfit.Parameter("sigma",       1, 0.1, 10)
lambd = zfit.Parameter("lambda",     -1,  -5,  0)
frac =  zfit.Parameter("fraction", 0.5,   0,  1)

gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
exponential = zfit.pdf.Exponential(lambd, obs=obs)
```

} parameters

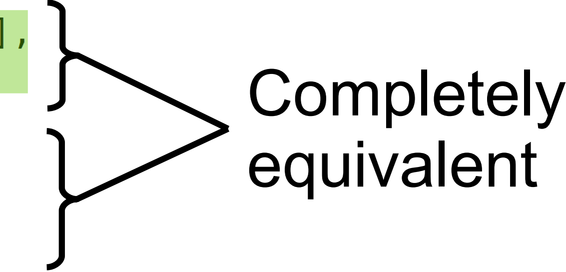
} models

Simultaneous fit

```
mu_shared = zfit.Parameter("mu_shared", 1., -4, 6)
sigma1 = zfit.Parameter("sigma_one", 1., 0.1, 10)
sigma2 = zfit.Parameter("sigma_two", 1., 0.1, 10)

gauss1 = zfit.pdf.Gauss(mu=mu_shared, sigma=sigma1, obs=obs)
gauss2 = zfit.pdf.Gauss(mu=mu_shared, sigma=sigma2, obs=obs)
```

shared parameters

```
nll_simultaneous = zfit.loss.UnbinnedNLL(model=[gauss1, gauss2],
                                          data=[data1, data2])

nll1 = zfit.loss.UnbinnedNLL(model=gauss1, data=data1)
nll2 = zfit.loss.UnbinnedNLL(model=gauss2, data=data2)
nll_simultaneous2 = nll1 + nll2
```



Completely equivalent