# Statistical inference & computational backends & statistics serialization

**Jonas Eschle**

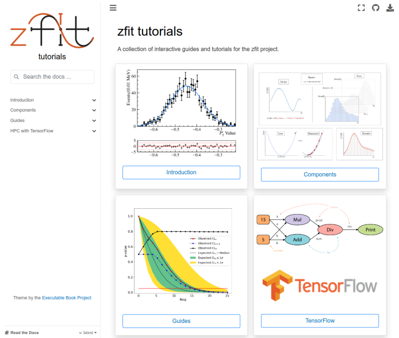
jonas.eschle@cern.ch

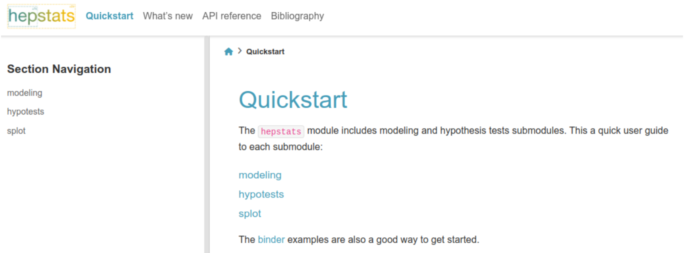
Swiss National Science Foundation

University of Zurich[UZH]

# About me

- Last months of PhD in experimental physics, LHCb, Zurich
  *from ~end of year post-doc in Syracuse, on zfit and friends*
- «By education, physics; by heart and skill, software & statistics»
- Since ~2018:
  - Main development of zfit
  - Dev of phasespace
  - Contributor (now maintainer) of hepstats
  - Maintainer (low) of formulate



# Outline

- Fitting landscape
- Computational backends
- Human readable serialization, HS3

# Outline

- Fitting landscape
- Computational backends
- Human readable serialization, HS3

Favouring hand-waving arguments/outdated knowledge
for a broader overview

# A brief history

~ year 2018: a lot of small projects are around

- No Scikit-HEP yet

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

*But what is fitting?*

# Fitting in HEP

# Statistical inference

Point estimate
(minimize, MAP)

Data

Bayesian CI

Model/PDF/(distribution)

Likelihood/loss/cost

Frequentist CI

Toys/asymptotic

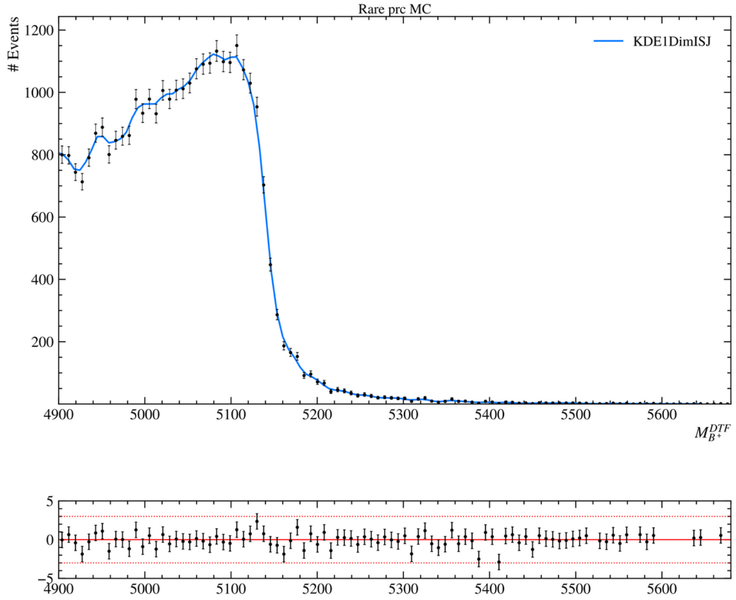
Frequentist Limits

# Different kind of fits

- Binned (*vs histfactory*) vs unbinned
  - Refers to data, cost/loss/likelihood and PDF
  - Unbinned data: product of PDFs
  - Binned data: «counting experiments»
- Template vs analytic
  - Shape from (simulation) sample vs closed-form function
- Analytical vs numerical normalization
  - Bin or closed-form integral vs numerical

# Different kind of fits

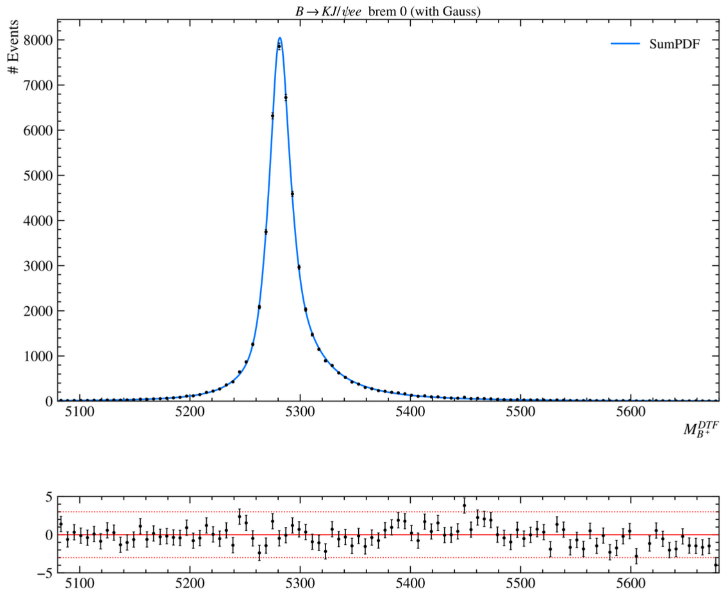
- Binned (*vs histfactory*) vs unbinned
  - Refers to data, cost/loss/likelihood and PDF
  - Unbinned data: product of PDFs
  - Binned data: «counting experiments»
- Template vs analytic
  - Shape from (simulation) sample vs closed-form function
- Analytical vs numerical normalization
  - Bin or closed-form integral vs numerical




**KDE**
Gaussian kernel → analytic norm
ISJ → numeric norm

# Different kind of fits

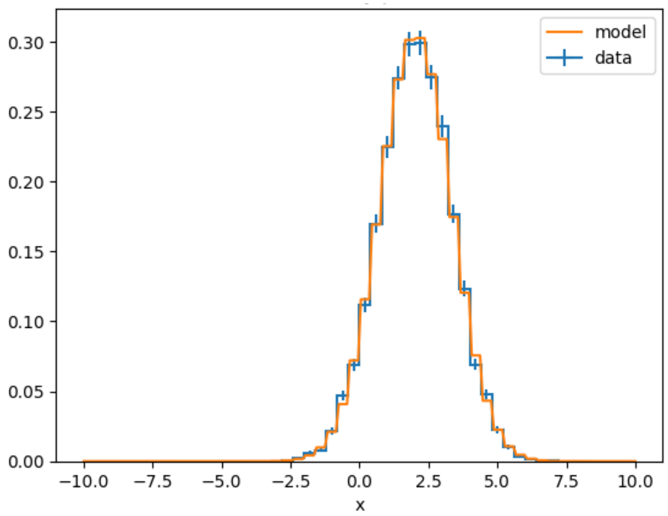
- Binned (*vs histfactory*) vs unbinned
  - Refers to data, cost/loss/likelihood and PDF
  - Unbinned data: product of PDFs
  - Binned data: «counting experiments»
- Template vs analytic
  - Shape from (simulation) sample vs closed-form function
- Analytical vs numerical normalization
  - Bin or closed-form integral vs numerical




**Double CB**

# Different kind of fits

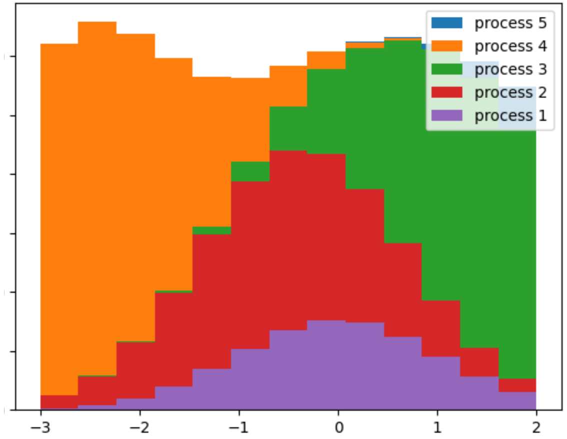
- Binned (*vs histfactory*) vs unbinned
  - Refers to data, cost/loss/likelihood and PDF
  - Unbinned data: product of PDFs
  - Binned data: «counting experiments»
- Template vs analytic
  - Shape from (simulation) sample vs closed-form function
- Analytical vs numerical normalization
  - Bin or closed-form integral vs numerical




**(binned) Gaussian fit to histogram**

# Different kind of fits

- Binned (*vs histfactory*) vs unbinned
  - Refers to data, cost/loss/likelihood and PDF
  - Unbinned data: product of PDFs
  - Binned data: «counting experiments»
- Template vs analytic
  - Shape from (simulation) sample vs closed-form function
- Analytical vs numerical normalization
  - Bin or closed-form integral vs numerical




**Stacked histograms PDFs**

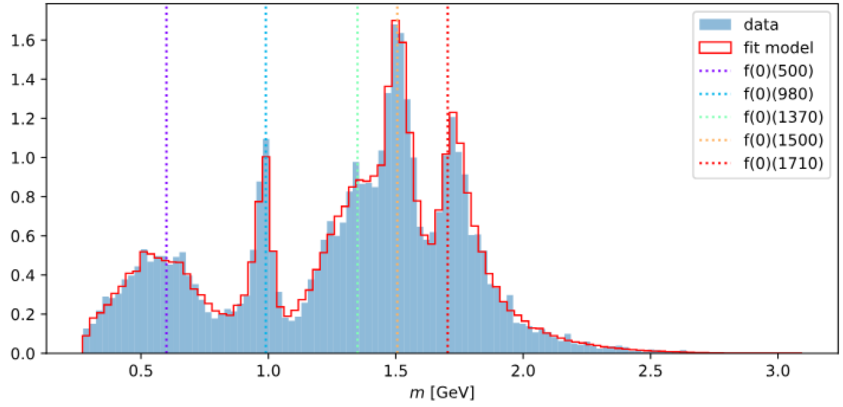
# pyhf-like models

- One extreme: HistFactory model (pyhf)
  - Template, binned, analytic normalization
  - Assumption: Bins «free-standing», not next to each other

- «Closed-world» fitter
  - Limited scope, specialized on 80%+ use-case in CMS/ATLAS
  - extremely powerful/tested, serializable

# Different kind of fits

- Binned (*vs histfactory*) vs unbinned
  - Refers to data, cost/loss/likelihood and PDF
  - Unbinned data: product of PDFs
  - Binned data: «counting experiments»
- Template vs analytic
  - Shape from (simulation) sample vs closed-form function
- Analytical vs numerical normalization
  - Bin or closed-form integral vs numerical




**Amplitude (partial wave) analysis**
**Angular analysis**

# Partial wave analysis

- The other extreme: amplitude analysis (ComPWA, …)
  - Unbinned, analytic, numerical normalisation
  - Description of observable based on amplitude, can be 1k + lines
- Fitting is also hard
  - Fitting time (~100 parameters): hours/days, up to weeks (one fit)
  - Bottleneck: evaluation of PDF

# Statistical inference landscape

Closed-world
HistFactory-like

pyhf
∂ifferentiable
ℒikelihoods

Point estimate
(minimize, MAP)

Open world
Binned,
unbinned,
mixed

Data

Model/PDF/(distribution)

Loss/Cost

Statistical inference

Boost Histogram

zfit

hepstats

iminuit

Building
amplitude
models

ComPWA

# Basic API example

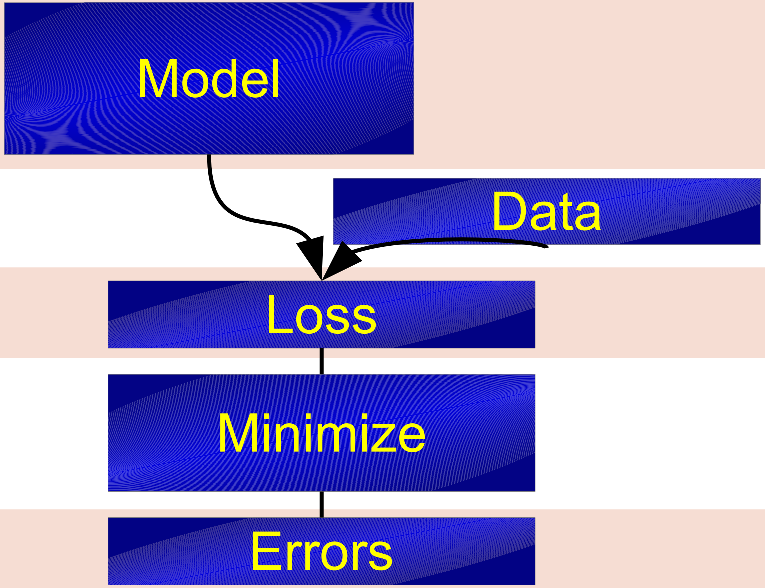
```
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
```

```
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()
```



# zfit features

- Extended fits, Chi2, binned, unbinned, mixed
- PDFs convertable binned ↔ unbinned (including to hist), mixed
- Multidimensional
- Any backend supported (numpy-like), optimal with TF currently
- Sample from PDF
- Arbitrary constraints (custom made)
- Custom PDF: define shape → auto normalized, sampling etc.
- Automatic/numerical gradient
- Different minimizers, optimized API
- JIT/eager support

# My take: fitting

- zfit, pyhf (also RooFit, HistFactory as C++ first) will co-exist
- API/Protocol needed in:
  - Fit parameters, data, variables (axis), distribution (.pdf, .integrate,…)

  … for
  - Plotting (mplhep?)
  - Hepstats?
- Hepstats can be more general
  *same interface that dispatches to two implementations?*
- *My job: zfit V2 (many things learnt)*

# Backends

# Backends overview

- Compiling vs tracing
  - Compile code (like cython, numba) to fast code
  - Trace computation «algebraic» (think Sympy), remember computation
- Gradient
  - Create «analytic» gradient from computations,
    apply chain rule consecutively
- Accelerators
  - Run on CPU, GPU, ...

# Backends compile

Numba, Cython

- Good for «event-by-event» computation
  - Event loop processing
- No gradient

# Backends trace

TensorFlow, JAX, *Sympy (converter to others)*

- Tracing with «algebraic» tensors
- (highly) optimized for vector computations
- Automatic gradients
- CPU, GPU, ...

# Detailed comparison

- TF, JAX vs Sympy
  - Sympy has algebraic knowledge, can do more powerful transformations

    ...but lacks the ability to do «loop-like», numerical things
  - Sympy can convert to JAX, TF etc
- TF vs JAX
  - JAX compilation subset of TF: only statically known shapes
  - JAX has no globals (but that’s maybe a good thing),
    but wide support for arbitrary object pass-through (pytree)
  - JAX has better support for arbitrary AD

# Cutting edge mentions

- Aesara (fork of Theano), backend of PyMC
  - Converts Sympy to JAX (and others) with optimizations
- Keras has now backend that supports multiple backends
- Data-api standard

# My take on backends

- Sympy (+ Aesara) to JAX seems promising
- JAX as the general choice
  - Sometimes less is more: multi-backend means also ***subset*** of features!
  - Crucial for more elaborate tasks like loops etc (numerical integrals)
- JIT if we can
- AD if we can

Requires communication standards for JIT & gradients

# Serialization

# HS3

## HEP Statistics Serialization Standard

*Human-readable & preservable format for HEP statistics*

- Serialize likelihood (including model, param, data, …)
- By RooFit, zfit and pyhf (+ more, growing), developing stage
- Explore and define common ground
  - What is a Gaussian/Gauss/Normal? Sum? Variable?

# HS3 goals

1) Publish and preserve
2) Create fit from scratch/edit existing
3) Exchange between libraries

Best effort base: «What works for all, works»

```
'pdfs': {'SumPDF': {'pdfs': [{'extended': 'n_sig',
                              'mu': 'mu',
                              'sigma': 'sigma',
                              'type': 'Gauss',
                              'x': 'x'},
                             {'extended': 'n_bkg',
                              'lam': 'lambda',
                              'type': 'Exponential',
                              'x': 'x'}],
                    'type': 'SumPDF'}},
'variables': {'lambda': {'max': -0.009999999776482582,
                         'min': -1.0,
                         'name': 'lambda',
                         'step_size': 0.001,
                         'value': -0.06294756382703781},
```

# hepstats

- Can serialize toy studies to yaml
  - Load toys instead of regenerating
  - Uses asdf, mixing yaml with binary

- Goal: move to/create HS3 inference standard


```yaml
toys:
- bestfit: !core/ndarray-1.0.0
    source: 0
    datatype: float64
    byteorder: little
    shape: [600]
  evalvalues: !core/ndarray-1.0.0
    source: 4
    datatype: float64
    byteorder: little
    shape: [2]
  genvalue: -0.09188308933186884
  nlls:
    -0.09188308933186884: !core/ndarray-1.0.0
      source: 1
      datatype: float64
      byteorder: little
      shape: [600]
    0.0: !core/ndarray-1.0.0
      source: 2
      datatype: float64
      byteorder: little
      shape: [600]
    bestfit: !core/ndarray-1.0.0
      source: 3
      datatype: float64
      byteorder: little
      shape: [600]
```

# Serialization — my take

- Parallel developement of «sub-formats»
- Needs «high-level-languages»: pyhf, amplitude analysis (physics)
- Best-effort base:
  - Library can (and should!) extend, go beyond standard
  - It should in the best case improve things, but never limit a library
- Challenges:
  - Store data (asdf file format? YAML with «auto hdf5 feature»), hist
  - Defining common statistical terms

# Summary

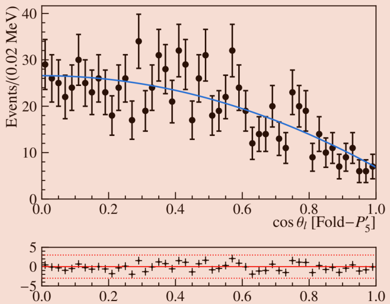
- Fitting landscape
- Computational backends
- Human readable serialization, HS3

Looking forward to discussions

*Bonus*

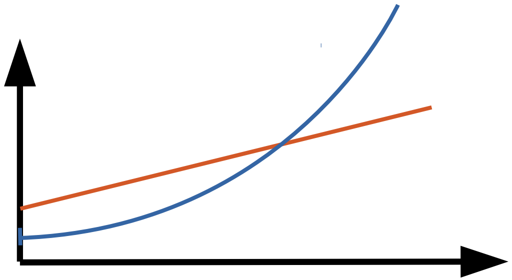
# Fitting with zfit

# HEP Model Fitting in Python




HEP

advanced features, simply extendable



Scalable

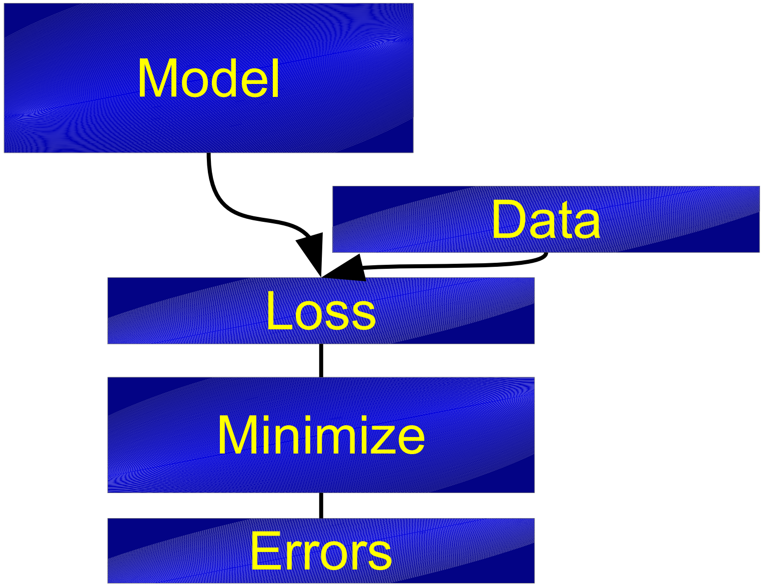
large data, complex models

Pythonic

integrate into ecosystem, stable API

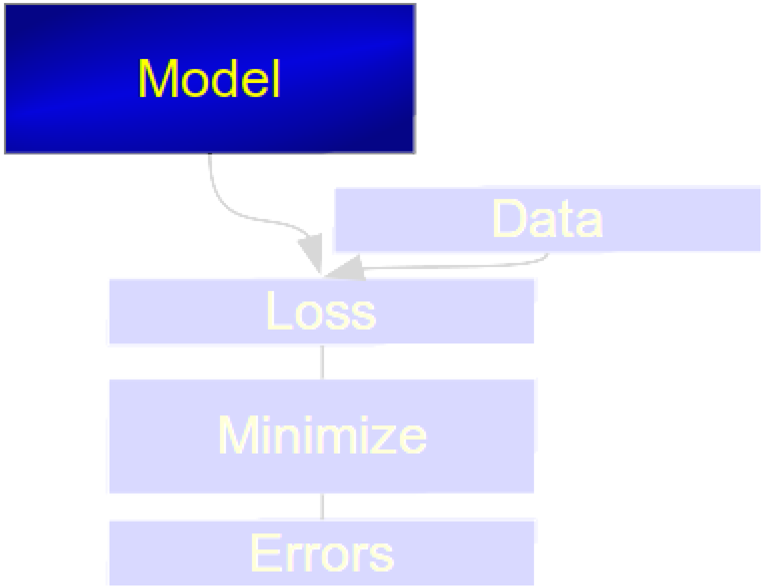
# Complete fit

```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()
```



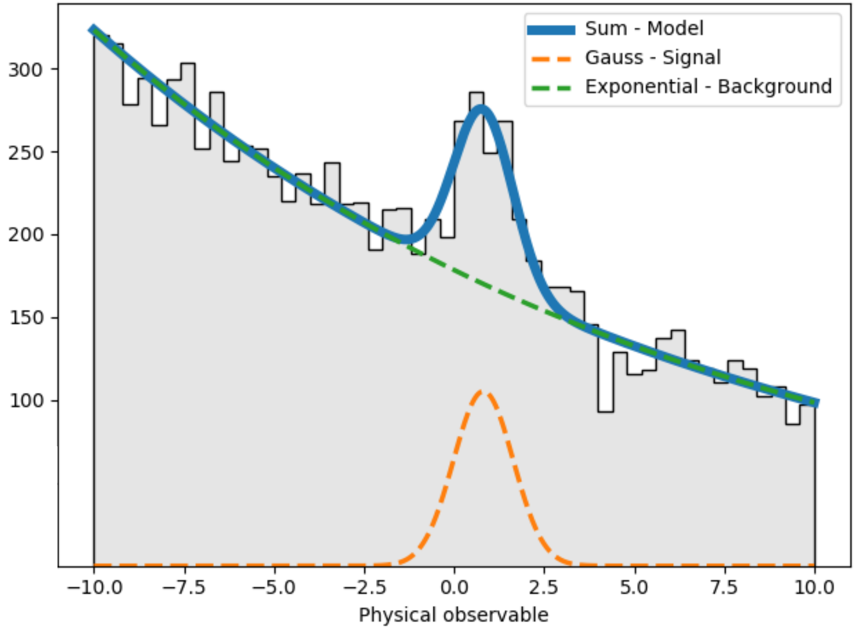
# Complete fit: Model

```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()
```



# 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", 0.5, 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

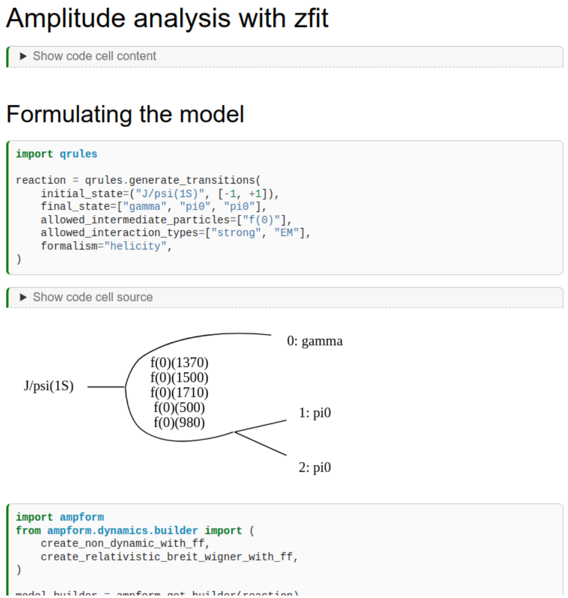
```python
custom_pdf = CustomPDF(obs=obs, alpha=0.2)

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

use functionality of model

# Arbitrary analytic shapes

```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
```

For example, create amplitude with ComPWA and zfit



Amplitude analysis with zfit

► Show code cell content

Formulating the model

```python
import qrules

reaction = qrules.generate_transitions(
    initial_state=("J/psi(1S)", [-1, +1]),
    final_state=["gamma", "pi0", "pi0"],
    allowed_intermediate_particles=["f(0)"],
    allowed_interaction_types=["strong", "EM"],
    formalism="helicity",
)
```

► Show code cell source



```python
import ampform
from ampform.dynamics.builder import (
    create_non_dynamic_with_ff,
    create_relativistic_breit_wigner_with_ff,
)
```

# Binned models

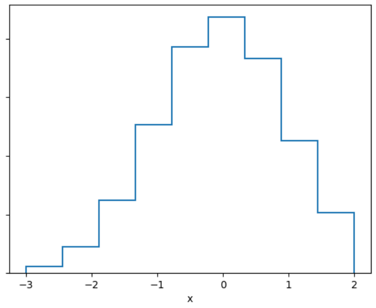
- Modelled after and compatible with boost-histogram/hist/UHI
  - Axes, names, ….
- Have "counts" and "rel_counts" method (returns hist-like)

```
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)
```

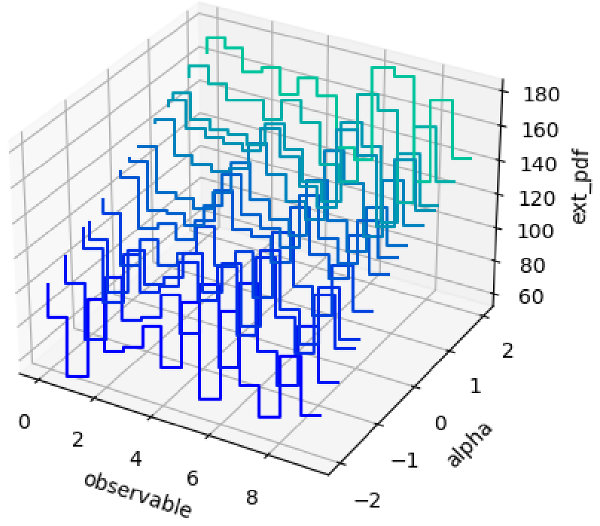
...and back

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

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

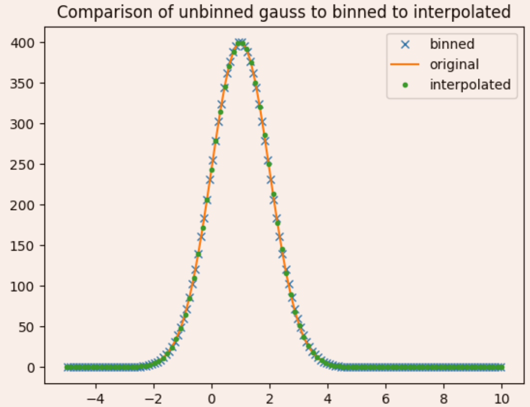


# More histograms



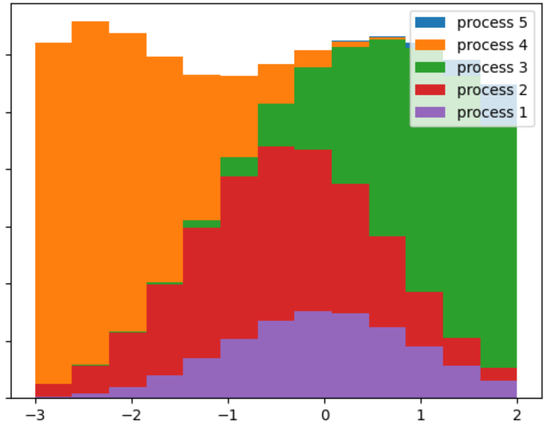

## Shape modifier

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




Unbinned → binned → interpolated

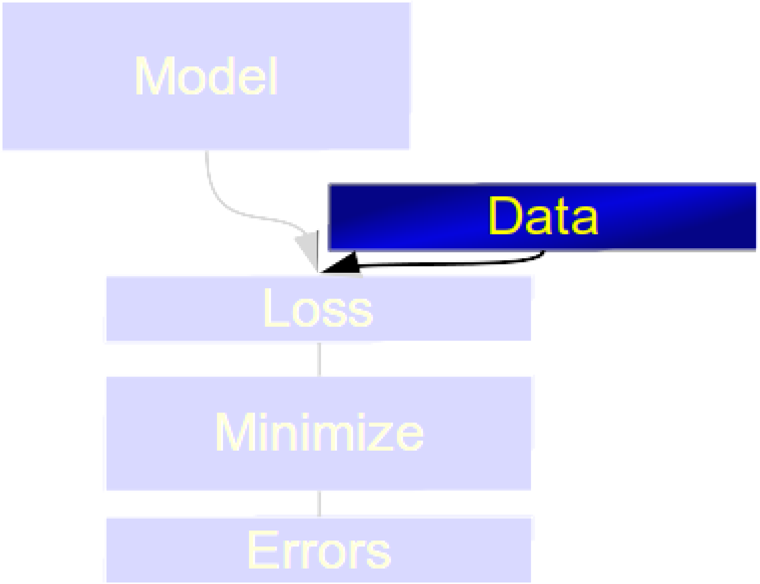
```
pdfs = [zfit.pdf.HistogramPDF(h) for h in histos]
alpha = zfit.Parameter('alpha', 0, -5, 5)
morph = SplineMorphingPDF(alpha=alpha, hists=pdfs)
```




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

# Complete fit: Data

```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

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

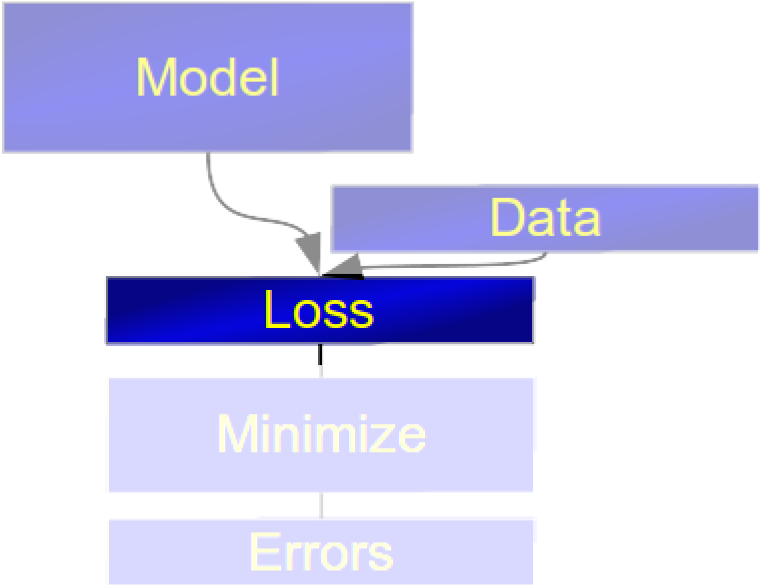
Use the HEP/Python ecosystem for preprocessing

- Sampled from a model (toy studies)

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

# 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

```python
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

```python
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
```

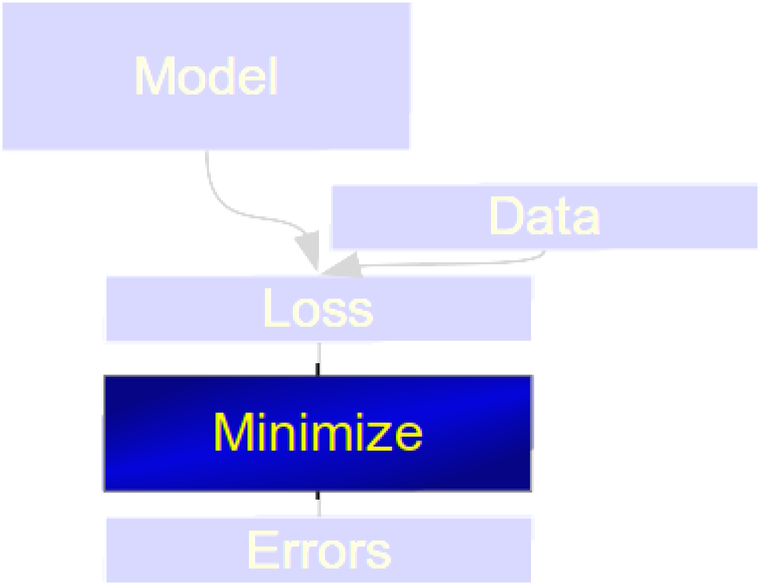
Equivalent

(arbitrary) constraints supported, added to loss

```python
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

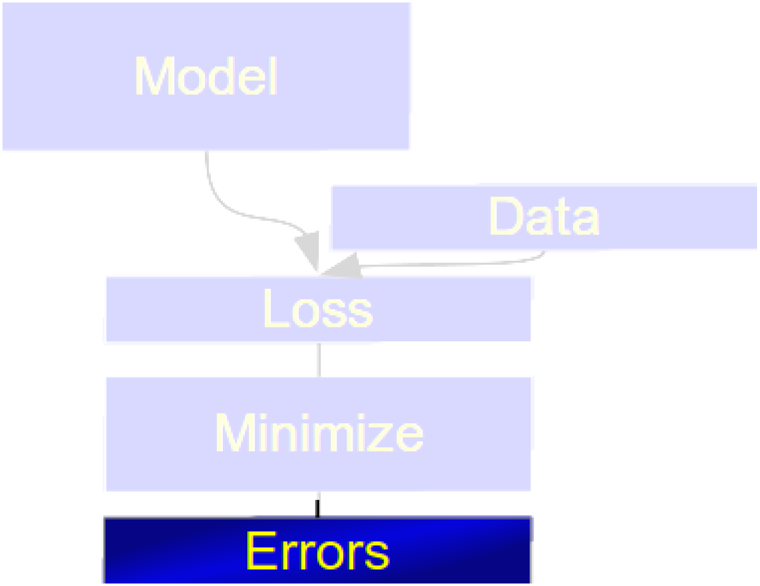
```
minimizer = zfit.minimize.IpyoptV1()
minimizer = zfit.minimize.Minuit()
minimizer = zfit.minimize.ScipyTrustConstrV1()
minimizer = zfit.minimize.NLoptLBFGSV1()
```

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

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

# Complete fit: Result

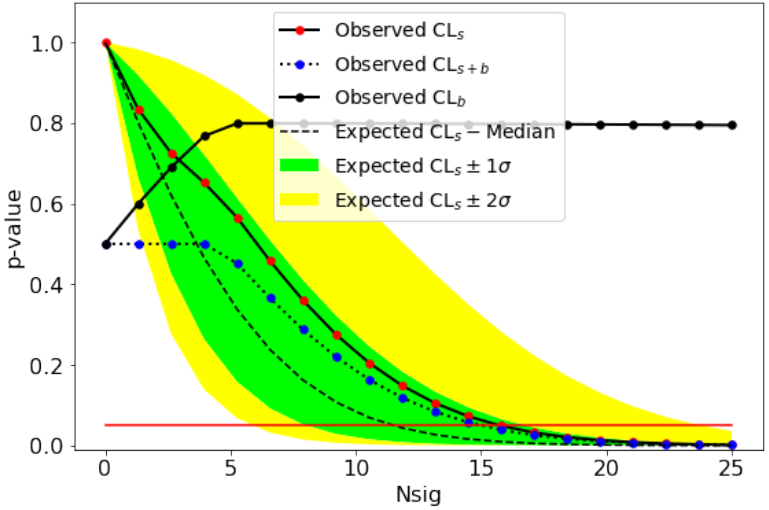
```
normal_np = np.random.normal(2., 3., size=10_000)

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()
```



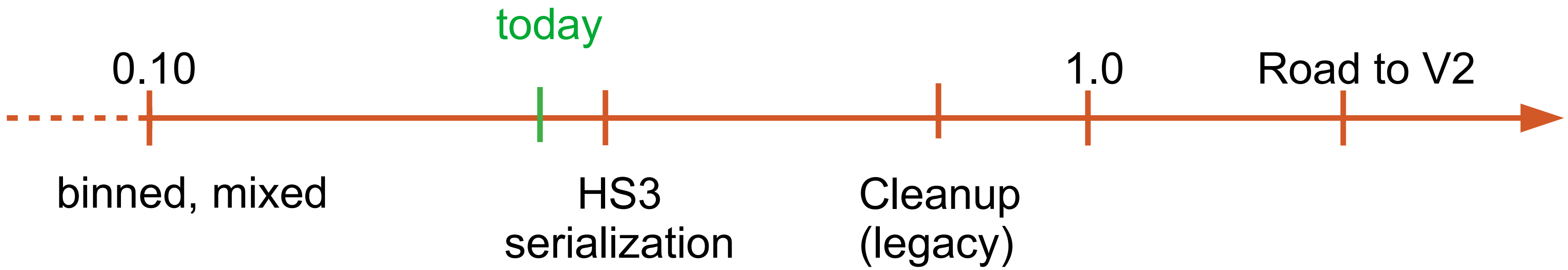
# Back to HEP ecosystem: hepstats

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

```
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




Lots of experience and proven API, but also design flaws (global parameters, ...)

Continue to incorporate feedback and adaptability to other libraries

V2 goal: incorporate other (smaller) fitting projects and have final API design