

# zfit: general likelihood model fitting in python



**Iason Krommydas (CMS)**

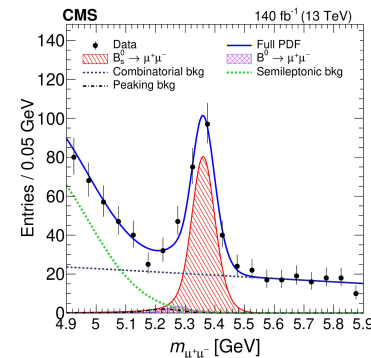
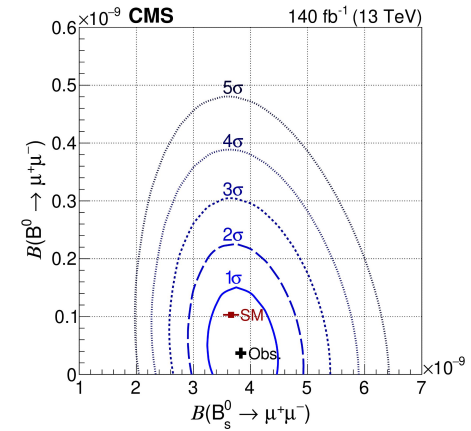
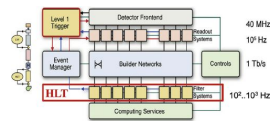
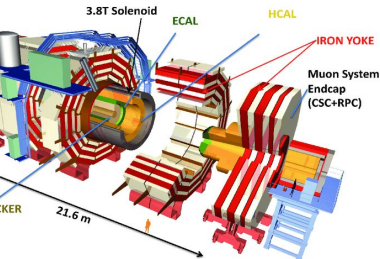
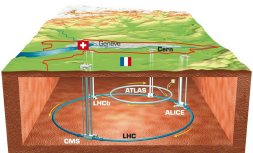
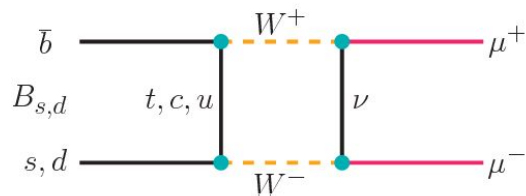
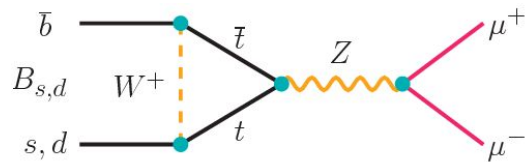
[ik23@rice.edu](mailto:ik23@rice.edu)

On behalf of the zfit team

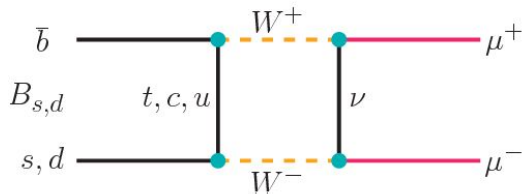
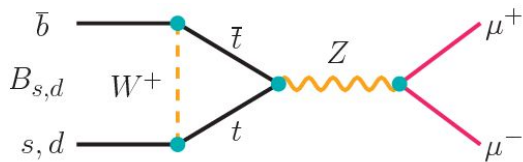


RICE UNIVERSITY

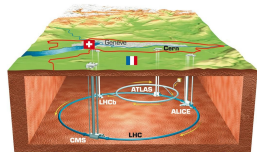
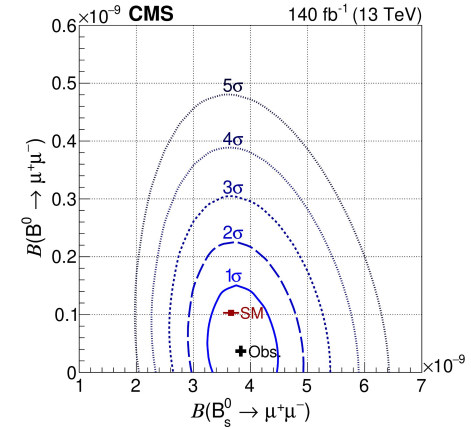
# HEP Analysis



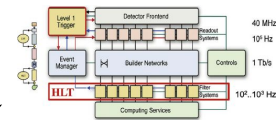
# HEP Analysis



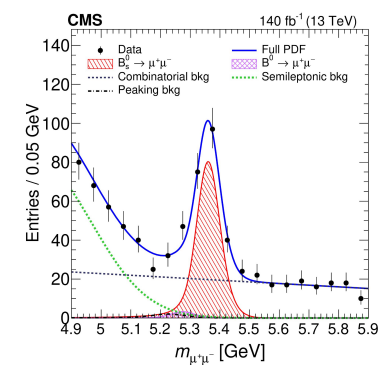
Lots of code



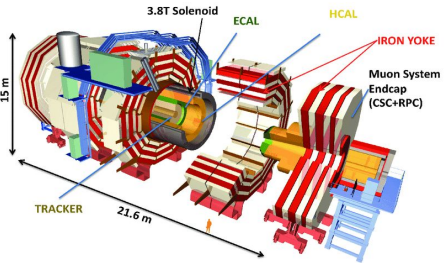
Lots of code



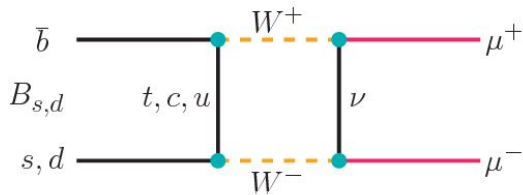
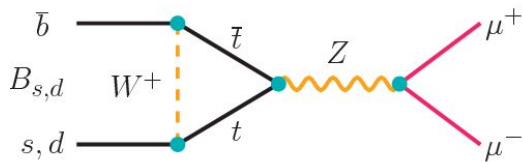
Lots of code



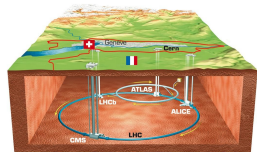
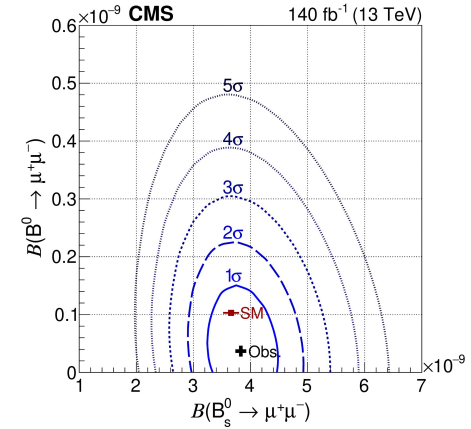
Lots of code



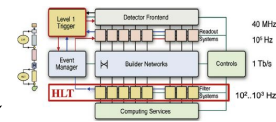
# HEP Analysis



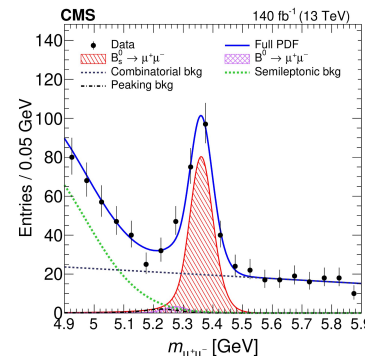
Lots of code



Lots of code



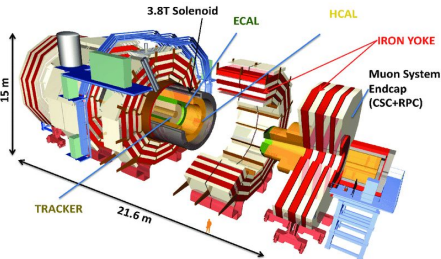
Lots of code



Lots of code

End-user analysis

Focus on «fitting» 4



# A brief history



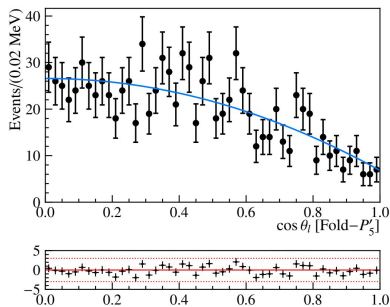
- A few years ago: analyses transition from C++ to Python in HEP
  - Scikit-HEP was created
  - Change of philosophy: non-monolithic packages
- Fitting packages still in in C++
  - Many scattered, specialized packages
  - Speed crucial aspect (and non-trivial in python)

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



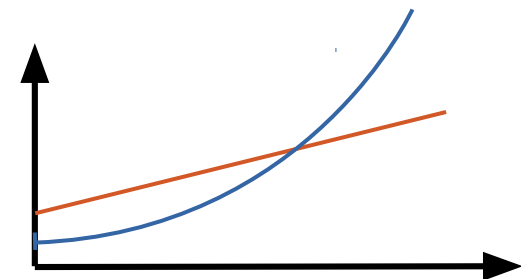
HEP

advanced features,  
simply extendable

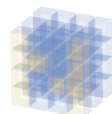


Scalable

large data, complex models



Pythonic



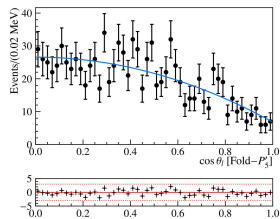
NumPy



python™

integrate into ecosystem, stable API

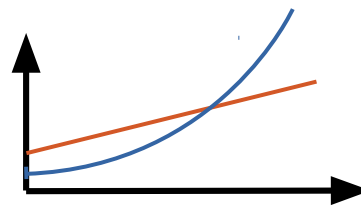
# HEP Model Fitting in Python



HEP  
advanced features,  
simply extendable



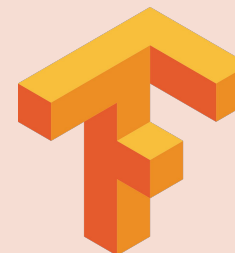
Scalable  
large data, complex models



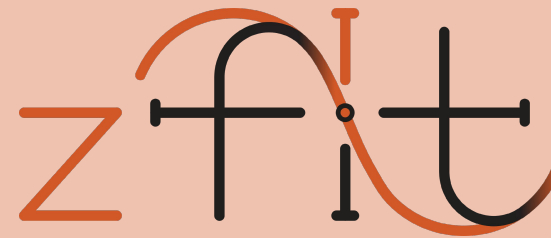
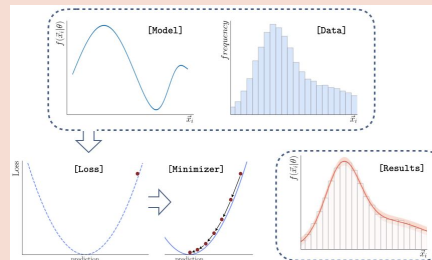
Pythonic  
integrate into ecosystem, stable API



Computing  
backend



API & Workflow





# Today

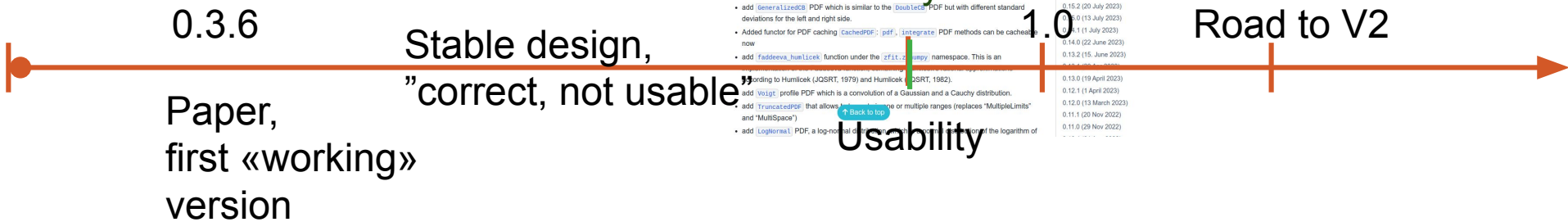


What's new? Getting started Tutorials API reference Project More

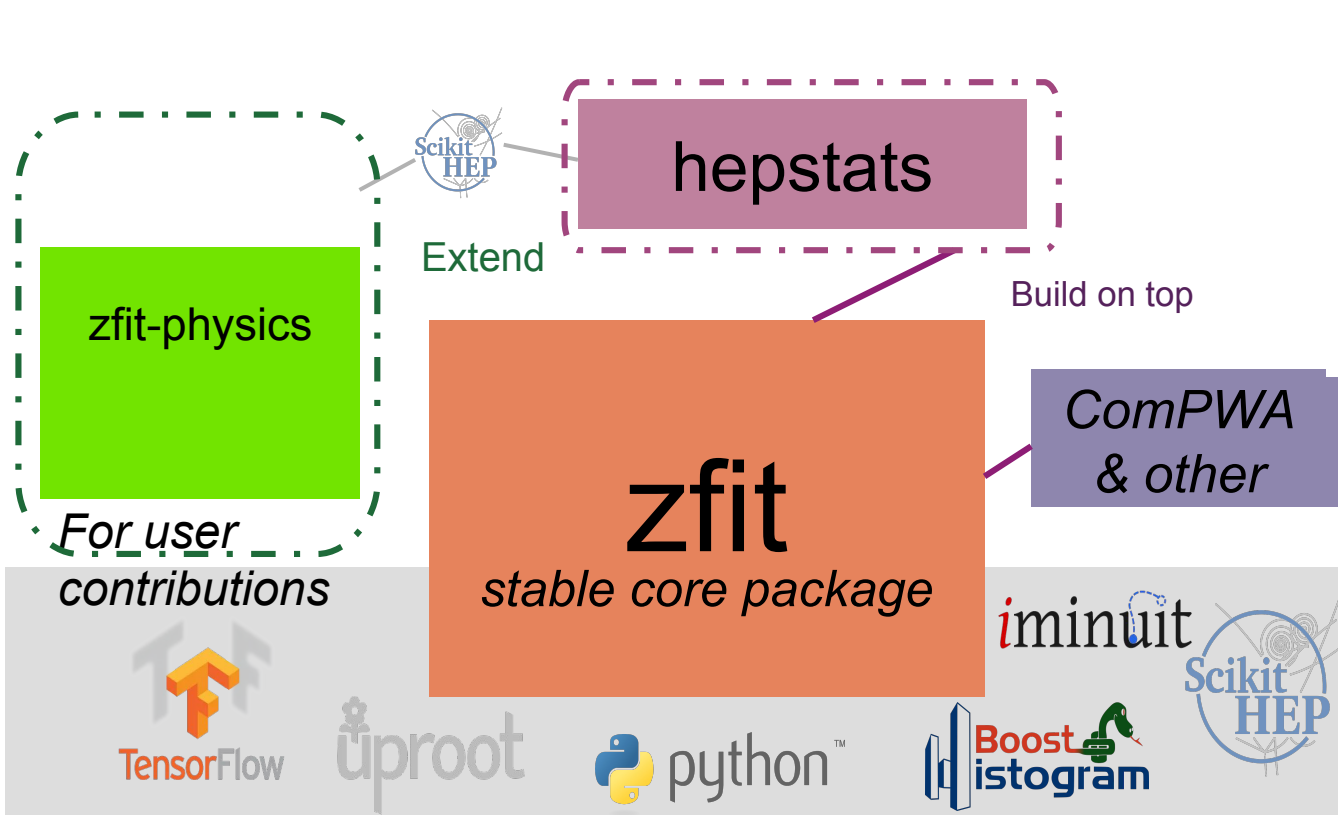
Complete overhaul of zfit with a focus on usability and a variety of new pdfs!

### Major Features and Improvements

- Parameter behavior has changed, multiple parameters with the same name can now coexist! The `UnboundLocalError` has been successfully removed (yay!). The new behavior only enforces that names and matching parameters within a `functionPDF` are unique, as otherwise inconsistent expectations appear (for the full discussion on this, see [here](#)).
- `Space` and `Limits` have a complete overhaul in front of them, in short, these overcomplicated objects get simplified and the `Limits` become more usable, in terms of dimensions. The full discussion and changes can be [found here](#).
- add an unbinned `Sampler` to the public namespace under `zfit.data.Sampler`; this object is returned in the `create_sampler` method and allows to resample from a function without recreating the compiled function, i.e. loss. It has an additional method `update_data` to update the data without recompiling the loss and can be created from a sample only. Useful to have a custom dataset in toys.
- allow to use pandas `DataFrame` as input where `zfit` Data objects are expected
- Methods of `PDFs` and loss functions that depend on parameters take now the value of a parameter explicitly as argument (instead of parameter name) to value.
- Python 3.12 support
- add `GeneralizedB` PDF which is similar to the `DoubleGauss` PDF but with different standard deviations for the left and right side.
- Added functor for PDF caching `cachedPDF`, `pdf`, `integrate` PDF methods can be cached now
- add `faddeeva_humlicek` function under the `zfit.functions.numpy` namespace. This is an `erfc` according to Humlicek (JQSRT, 1979) and Humlicek (JQSRT, 1982).
- add `Voigt` profile PDF which is a convolution of a Gaussian and a Cauchy distribution.
- add `TruncatedPDF` that allows for one or multiple ranges (replaces "MultipleLimits" and "MultiSpace")
- add `LogGamma` PDF, a log-normal distribution, which is the logarithm of the



# Ecosystem



# Fitting with zfit

# Complete fit



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

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

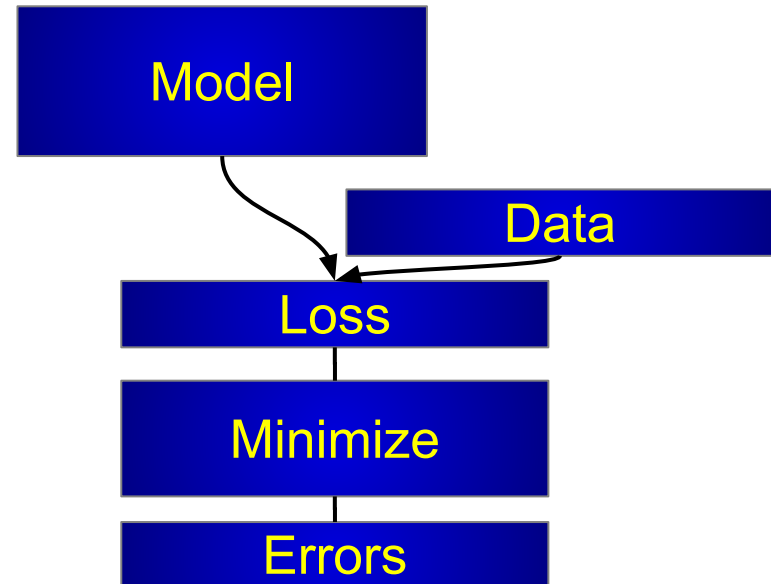
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

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))

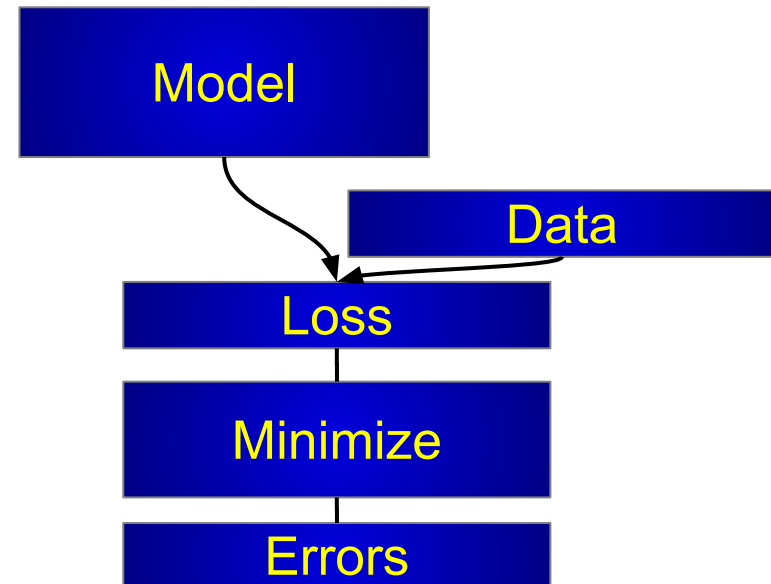
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

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()
```

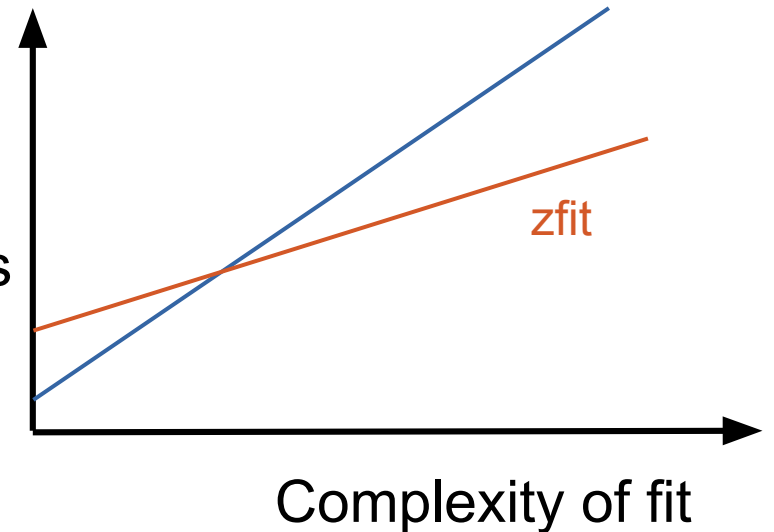


# Scalable: Usability



- *Things should not be easy or hard, but consistent*
- Code lines
  - 5 or 10: irrelevant
  - 50 or 300: matters
- Cover all use cases out of the box is impossible
  - Convenient base classes, allow full control
  - Modular structure; provide all elements (e.g. shapes)

Difficulty /  
# code lines



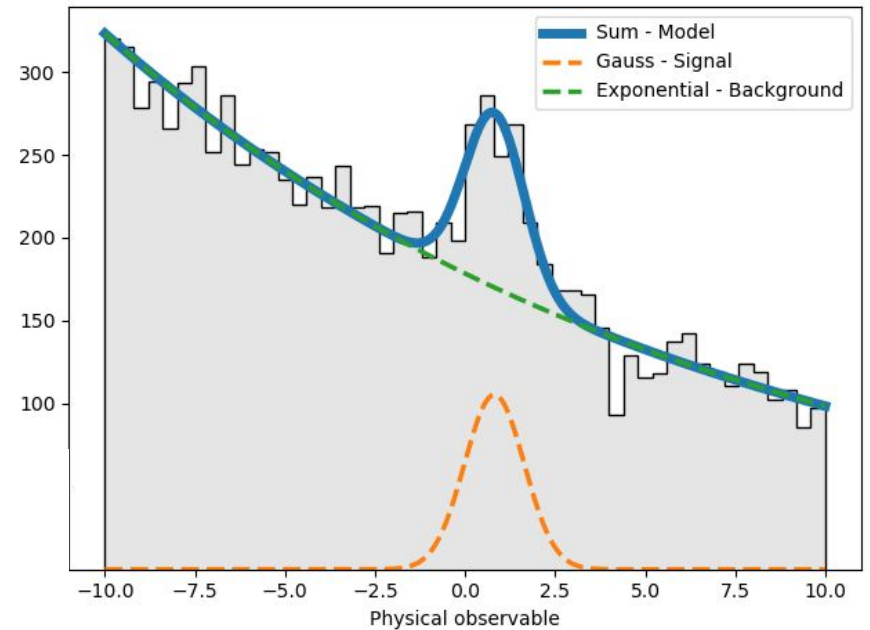
# Available models



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

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

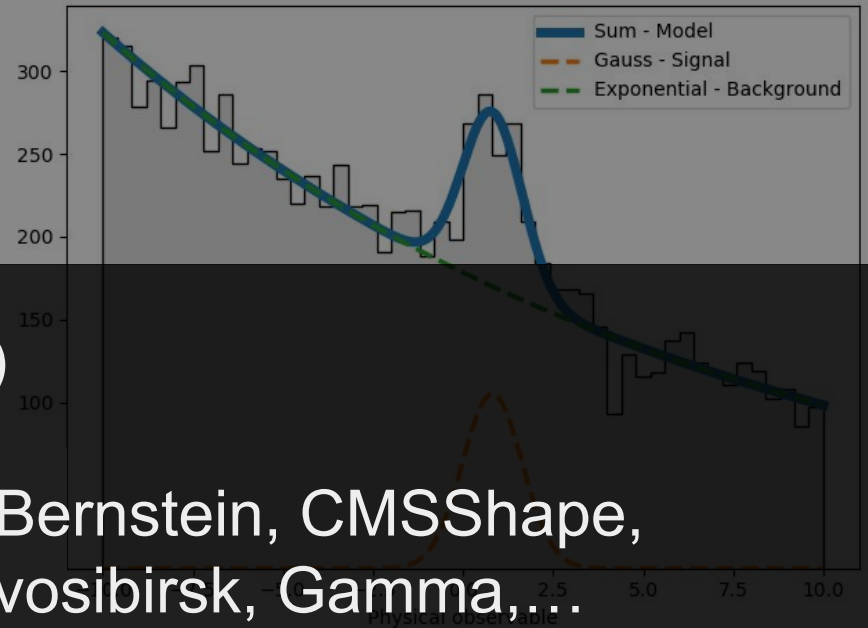
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
exponential = zfit.pdf.Exponential(lambda, obs=obs)
model = zfit.pdf.SumPDF([gauss, exponential], fracs=frac)
```



# Available models



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



A lot of new PDFs (also in zfit-physics)

```
lambda = zfit.Parameter("lambda", -0.06, -1, -0.01)
gauss = zfit.pdf.Gauss("Gauss", lambda, obs=obs)
exponential = zfit.pdf.Exponential(lambda, obs=obs)
model = zfit.pdf.SumPDF([gauss, exponential], fracs=frac)
```

Voigt, GeneralizedCB, GaussExpTail, Bernstein, CMSShape, LogNormal, Cruijff, TruncatedPDF, Novosibirsk, Gamma,...



## The simple way

While the same works for functions, an example with a PDF is shown here.

```
import numpy as np
import zfit
from zfit import z
```

The first way is the most simple and should only be used for the trivial cases, i.e. if you're not familiar with Python classes (especially not with the `__init__` method).

```
class MyGauss(zfit.pdf.ZPDF):
    _N_OBS = 1 # dimension, can be omitted
    _PARAMS = ['mean', 'std'] # the name of the parameters

    @zfit.supports()
    def _unnormalized_pdf(self, x, params):
        x0 = x[0] # using the 0th axis
        mean = params['mean']
        std = params['std']
        return z.exp(-((x0 - mean) / std) ** 2)
```

Done. Now we can use our pdf already!

The slightly more general way involves overwriting the `__init__` and gives you all the possible flexibility: to use custom parameters, to preprocess them etc.

Here we inherit from `BasePDF`

```
class MyGauss(zfit.pdf.BasePDF):

    def __init__(self, mean, std, obs, extended=None, norm=None, name=None, label=None):
        params = {'mean': mean, # 'mean' is the name as it will be named in the PDF, mean
                  'std': std
                 }
        super().__init__(obs=obs, params=params, extended=extended, norm=norm,
                        name=name, label=label)

    @zfit.supports()
    def _unnormalized_pdf(self, x, params):
        x0 = x[0] # using the 0th axis
        mean = params['mean']
        std = params['std']
        return z.exp(-((x0 - mean) / std) ** 2)
```

Can also override:

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

Or register integral

# Custom PDF



```
obs = zfit.Space('obs1', -3, 6)

data_np = np.random.random(size=1000)
data = zfit.Data(data_np, obs=obs)
```

Create two parameters and an instance of your own pdf

```
mean = zfit.Parameter("mean", 1.)
std = zfit.Parameter("std", 1.)
my_gauss = MyGauss(obs=obs, mean=mean, std=std)
```

```
probs = my_gauss.pdf(data)
```

```
print(probs[:20])
```

```
tf.Tensor(
[0.44462038 0.56276035 0.35691916 0.37072295 0.51695323 0.20865716
 0.28003744 0.42683103 0.44741831 0.55557742 0.51748377 0.49276503
 0.47680781 0.30154858 0.52949263 0.2574051 0.31240842 0.31386275
 0.28662323 0.36839776], shape=(20,), dtype=float64)
```

We could improve our PDF by registering an integral

```
def gauss_integral_from_any_to_any(limits, params, model):
    lower, upper = limits.v1.limits
    mean = params['mean']
    std = params['std']
    # write your integral here
    return 42. # dummy integral, must be a scalar!
```

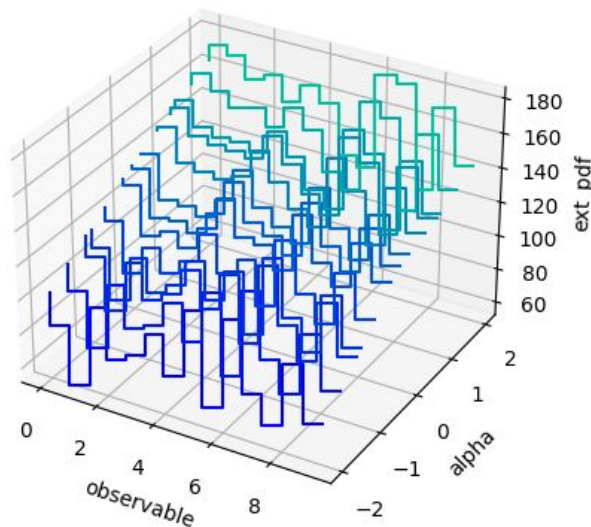
```
limits = zfit.Space(axes=0, lower=zfit.Space.ANY_LOWER, upper=zfit.Space.ANY_UPPER)
MyGauss.register_analytic_integral(func=gauss_integral_from_any_to_any, limits=limits)
```

Can also override:

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

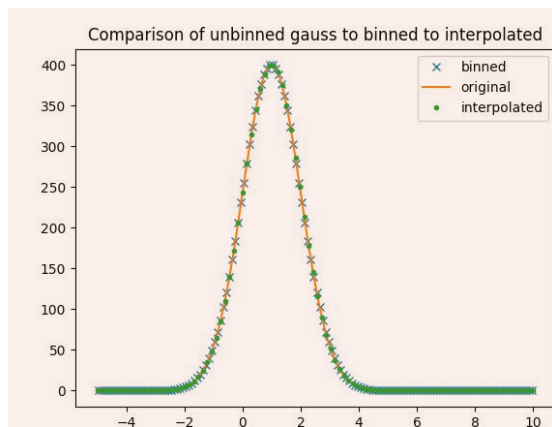
Or register integral

# Binned PDFs

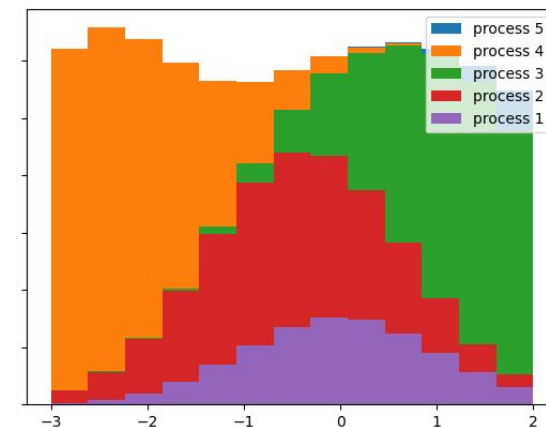


## Shape modifier

```
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, hist=pdfs)
```

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

# Binned PDFs



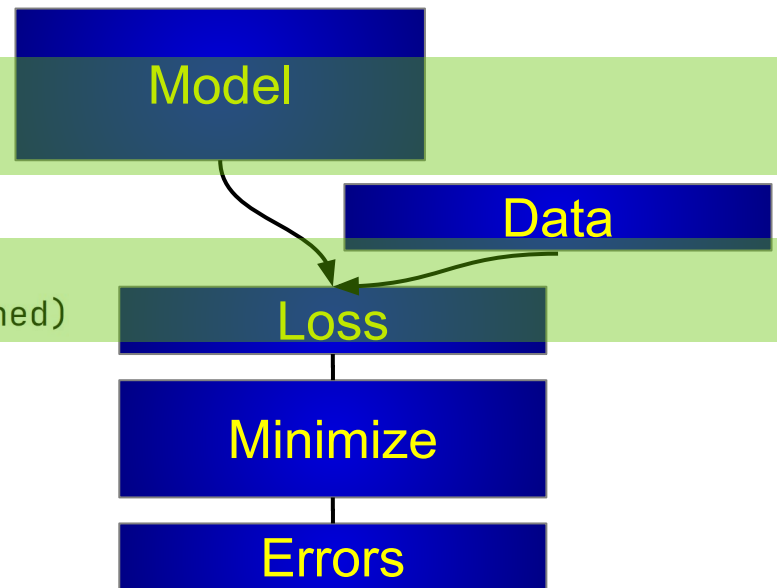
## Going binned

```
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 = gauss.to_binned(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()
```



# Complete fit: Data



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

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

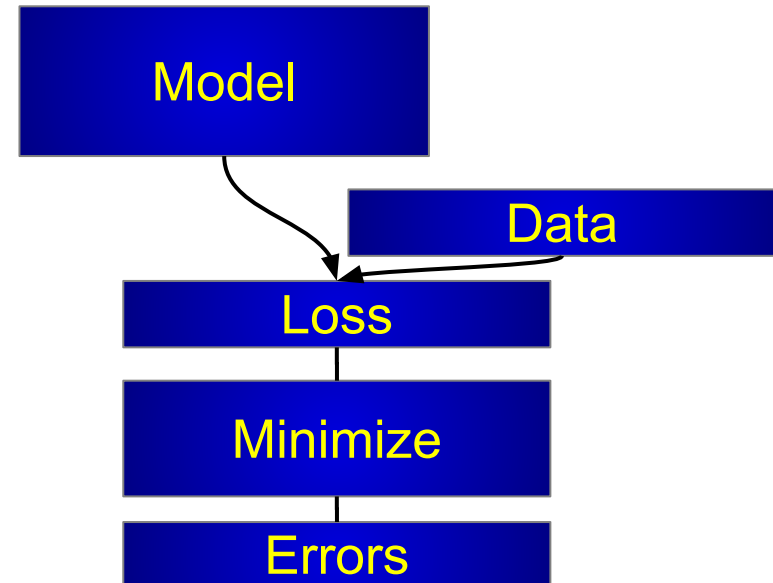
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

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)
```

# Complete fit: Loss



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

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

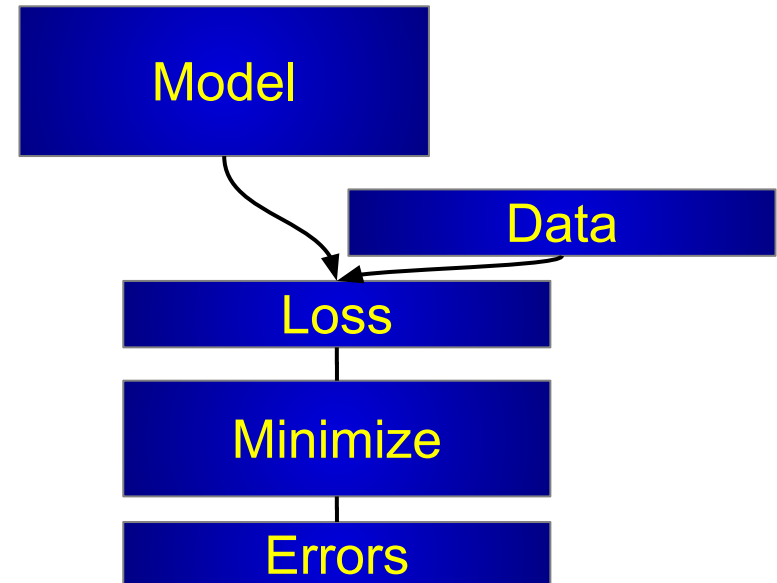
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

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
```

} Equivalent

(arbitrary) constraints supported, 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))

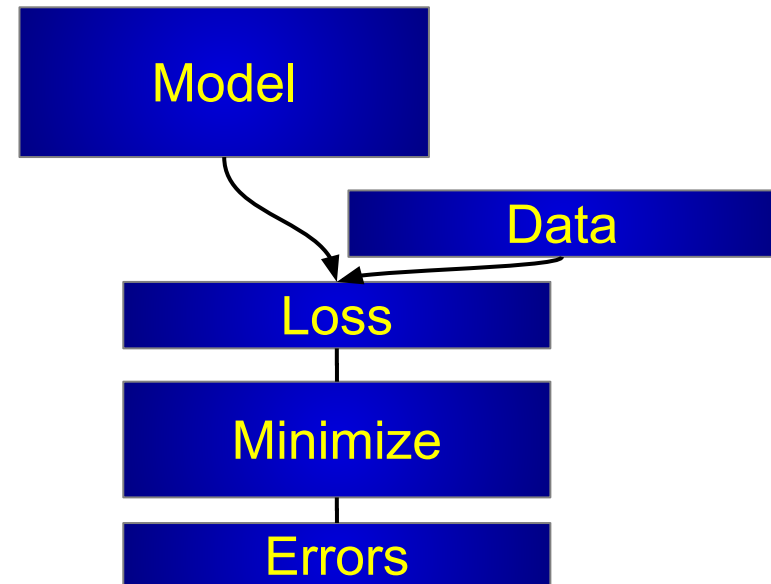
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

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 interface "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.NLoptLBFSGSV1()
```

- 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))

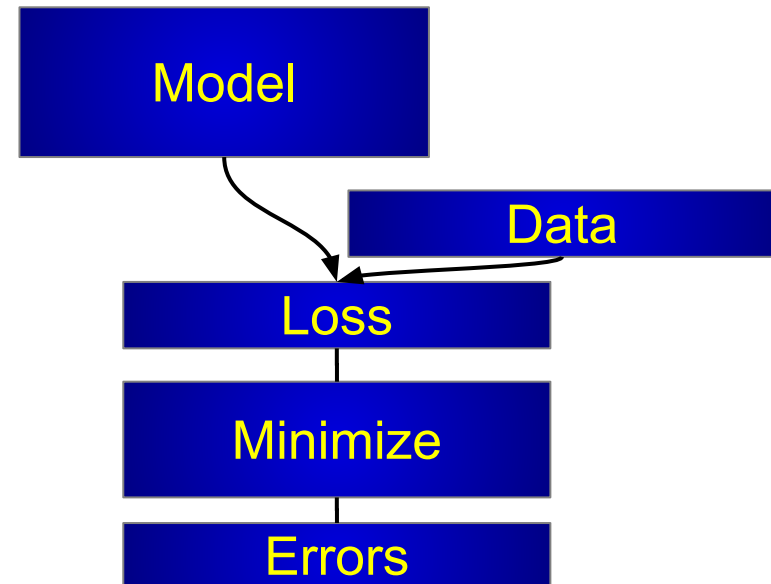
mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.5, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

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()
```



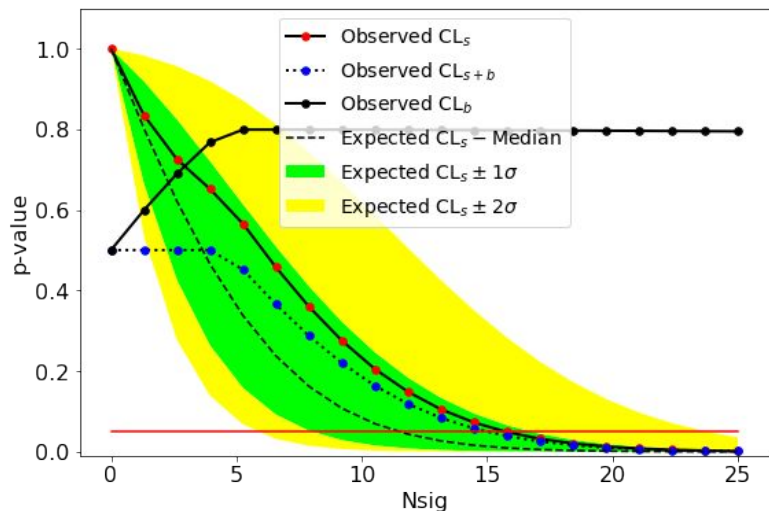
# Back to HEP ecosystem: hepstats



- 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)
```



**How to preserve this?**

- Longstanding goal: preserve and restore information
- Question: how? Which format?
  - Human-readable vs binary, ...
- zfit: Pickle dump/load objects (PDF, etc)

**Massimo Corradi**

It seems to me that there is a general consensus that what is really meaningful for an experiment is *likelihood*, and almost everybody would agree on the prescription that experiments should give their likelihood function for these kinds of results. Does everybody agree on this statement, to publish likelihoods?

**Louis Lyons**

Any disagreement? Carried unanimously. That's actually quite an achievement for this Workshop.

(1st Workshop on Confidence Limits, CERN, 2000)

- HEP Statistics Serialization Standard
  - Human-readable & preservable format for HEP statistics*
- By RooFit, zfit, pyhf, bat.jl, ... developing stage
  - Explore and define common ground
- zfit: Can dump/load (some) PDFs HS<sup>3</sup>-like


```
'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},
```

# Backend & Tensorflow



# Main backend: TensorFlow



- By Google, highly popular (190k★, top 17 on )
- 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)
  - JIT compiled (very performant)



**GPU/Multi CPU support**  
**Automatic Gradient**

# Backend: tracing and autograd



Tracing (JIT compilation)

*execute Python once, remember (algebraic) computation*

Autograd

*"analytic" gradient of function*



Recent rise of big data industry created libraries that support this

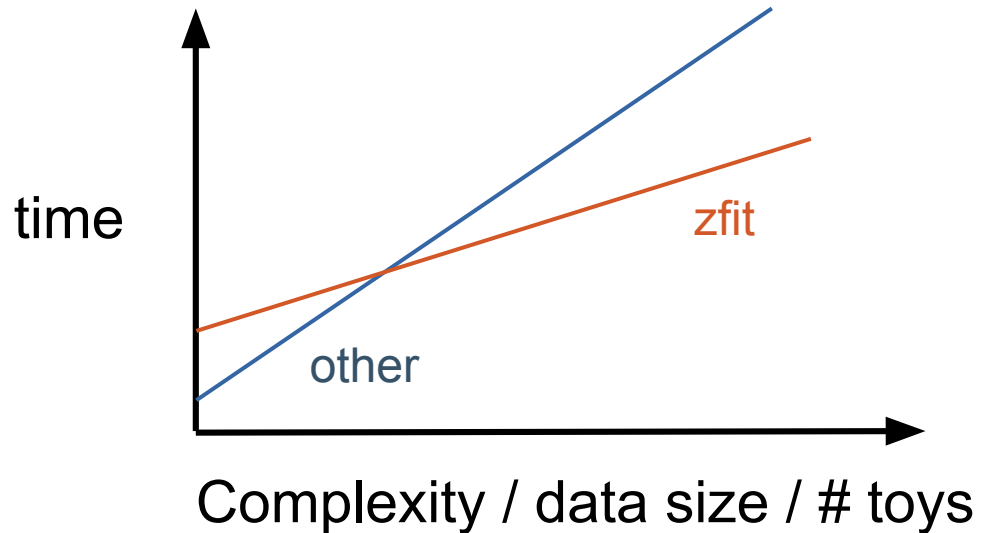
Includes GPU support, optimizations, caching,...

JAX in the future for zfit2

# 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



Same-ish order of magnitude as RooFit

# Summary



- zfit has matured enough to provide a stable fitting package within the pythonic HEP ecosystem.
- Extensive feature set for model building, data loading, minimization and getting results.
- Can serialize objects and save/load them to/from disk.
- Robust backend that supports jit compilation, automatic differentiation, GPU support and scales well
- Good usability scaling.

Where to find it:

- [zfit repository](#)
- [zfit-physics repository](#)
- [zfit documentation](#)
- [zfit tutorials](#)
- [hepstats repository](#)
- [hepstats documentation](#)

# Backup Slides

# Statistical inference landscape

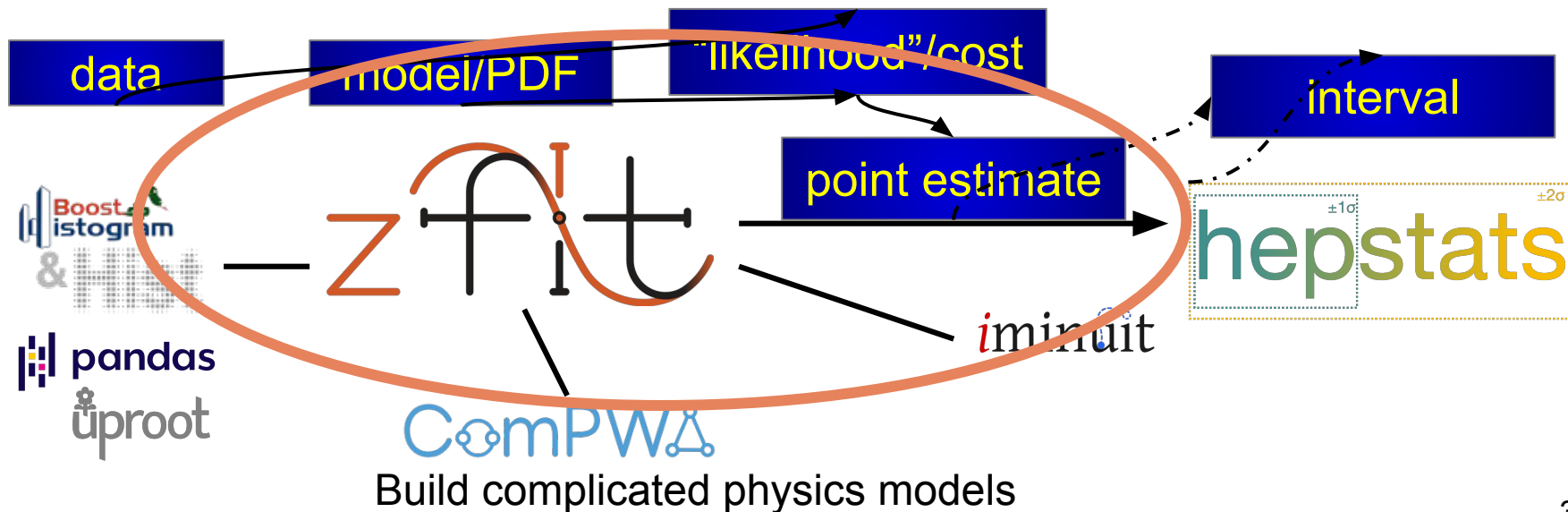


**cabinetry** Steers large fits & analysis

Closed-world  
HistFactory-like



Open-world  
Binned,  
unbinned,  
mixed



- Extended fits, Chi2, binned, unbinned, mixed
- PDFs convertible binned  $\leftrightarrow$  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  $\rightarrow$  auto normalized, sampling etc.
- Automatic/numerical gradient
- Different minimizers, optimized API
- JIT/eager support