dilax: differentiable (binned) likelihoods with JAX

[dila:x]

-

Peter Fackeldey

pyhf Workshop

07.12.23

RWTH AACHEN UNIVERSITY

dilax at a glance

What is *dilax*?

- Python library (*pip install dilax*) to define likelihoods and work with them
- Provides a statistical model definition similar to torch.nn.Module
- Fully based on JAX as backend
- Integrates nicely with JAX ecosystem (just examples):
  - Based on: JAX, Equinox
  - Optimizers: JAXopt, Optimistix, Optax
  - Utilities: chex, orbax

Key Concept:

- Everything in *dilax* is a JAX PyTree

  → fully compatible with JAX composable transformations: jax.jit, jax.grad, ...
- This key concept allows for:
  - jax.grad: gradients through likelihoods (like *neos*)
  - jax.jit: GPU acceleration, remove Python overhead, computing graph optimization through XLA
  - jax.vmap: simultaneously run thousands of fits on a GPU (likelihood profile), vectorize evaluation of different stat. Models, vectorize toy sampling, ...

Building blocks & components of *dilax*


```
import dilax as dlx
```

`jax.Array`

`dlx.Parameter`

`dlx.likelihood.NLL`

$$\mathscr{L} = \prod_i \text{Pois}(d_i, \lambda_i(s_i, b_i, \vec{\theta})) \cdot \prod_j \pi_j(\theta_j)$$

`dlx.Model(...).evaluate()`

`dlx.effect.{gauss,shape,lnN,poisson}`

Available 'Methods'

- Negative log-likelihood (+ profiling)
- Hessian, Covariance matrix
- Toy sampling
- CLs/limit calculation (WIP)

Negative log-likelihood fit

```python
import equinox as eqx
import jax
import jax.numpy as jnp

import dilax as dlx

jax.config.update("jax_enable_x64", True)


class MyModel(dlx.Model):
    def __call__(
        self, processes: dict, parameters: dict[str, dlx.Parameter]
    ) -> dlx.Result:
        ...


# setup model
model = MyModel(...)

# define negative log-likelihood with data (observation)
nll = dlx.likelihood.NLL(model=model, observation=jnp.array([...]))
fast_nll = eqx.filter_jit(nll)

# setup fit: initial values of parameters and a suitable optimizer
init_values = model.parameter_values
optimizer = dlx.optimizer.JaxOptimizer.make(
    name="ScipyMinimize", settings={"method": "trust-constr"}
)

# fit
fitted_params, state = optimizer.fit(fun=fast_nll, init_values=init_values)
```

Negative log-likelihood fit

```python
import equinox as eqx
import jax
import jax.numpy as jnp

import dilax as dlx

jax.config.update("jax_enable_x64", True)
```

Model

```python
class MyModel(dlx.Model):
    def __call__(
        self, processes: dict, parameters: dict[str, dlx.Parameter]
    ) -> dlx.Result:
        ...


# setup model
model = MyModel(...)
```

NLL

```python
# define negative log-likelihood with data (observation)
nll = dlx.likelihood.NLL(model=model, observation=jnp.array([...]))
fast_nll = eqx.filter_jit(nll)
```

Fit

```python
# setup fit: initial values of parameters and a suitable optimizer
init_values = model.parameter_values
optimizer = dlx.optimizer.JaxOptimizer.make(
    name="ScipyMinimize", settings={"method": "trust-constr"}
)

# fit
fitted_params, state = optimizer.fit(fun=fast_nll, init_values=init_values)
```

dilax Model example


```python
# define a simple model with two processes and two parameters
class MyModel(dlx.Model):
    def __call__(
        self, processes: dict, parameters: dict[str, dlx.Parameter]
    ) -> dlx.Result:
        res = dlx.Result()

        # signal
        mu_mod = dlx.modifier(
            name="mu", parameter=parameters["mu"], effect=dlx.effect.unconstrained()
        )
        res.add(process="signal", expectation=mu_mod(processes["signal"]))

        # background
        bkg_mod = dlx.modifier(
            name="sigma", parameter=parameters["sigma"], effect=dlx.effect.gauss(0.2)
        )
        res.add(process="background", expectation=bkg_mod(processes["background"]))
        return res


# setup model
processes = {"signal": jnp.array([10.0]), "background": jnp.array([50.0])}
parameters = {
    "mu": dlx.Parameter(value=jnp.array([1.0]), bounds=(0.0, jnp.inf)),
    "sigma": dlx.Parameter(value=jnp.array([0.0])),
}
model = MyModel(processes=processes, parameters=parameters)
```

Modifiers

- Parameters $\vec{\theta}$ modify bin contents (λ_i) of the expectation through an 'effect'
- Available effects: unconstrained, lnN, gauss, poisson, shape

```python
import jax.numpy as jnp
import dilax as dlx

mu = dlx.Parameter(value=1.1, bounds=(0, 100))
norm = dlx.Parameter(value=0.0, bounds=(-jnp.inf, jnp.inf))

# unconstrained effect
modify = dlx.modifier(name="mu", parameter=mu, effect=dlx.effect.unconstrained())

# apply the modifier
modify(jnp.array([10, 20, 30]))
# -> Array([11., 22., 33.], dtype=float32, weak_type=True),

# lnN effect
modify = dlx.modifier(name="norm", parameter=norm, effect=dlx.effect.lnN(0.2))

# poisson effect
hist = jnp.array([10, 20, 30])
modify = dlx.modifier(name="norm", parameter=norm, effect=dlx.effect.poisson(hist))

# shape effect
up = jnp.array([12, 23, 35])
down = jnp.array([8, 19, 26])
modify = dlx.modifier(name="norm", parameter=norm, effect=dlx.effect.shape(up, down))
```

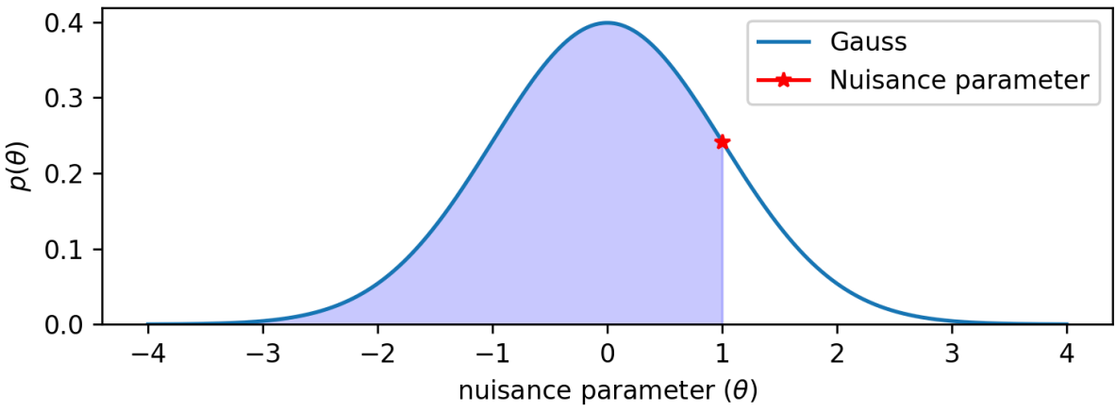
Fully factorized constraint concept

- Idea: Distinguish between the constraint term for the likelihood and the effect of the pdf that changes the expectation
- (Almost) every effect defines its constraint through a Gaussian with 0 mean and width of 1 ($\mathcal{G}(0,1)$)
- The translation between $\mathcal{G}(0,1)$ and the scale factor for the bin exp. of any effect can be calculated with the (inverse) CDFs:

$$\text{SF}(\theta) = \text{iCDF}(\pi(X))\Big[\text{CDF}(\mathcal{G}(0,1))(\theta)\Big] \qquad \text{(Eq. 1)}$$

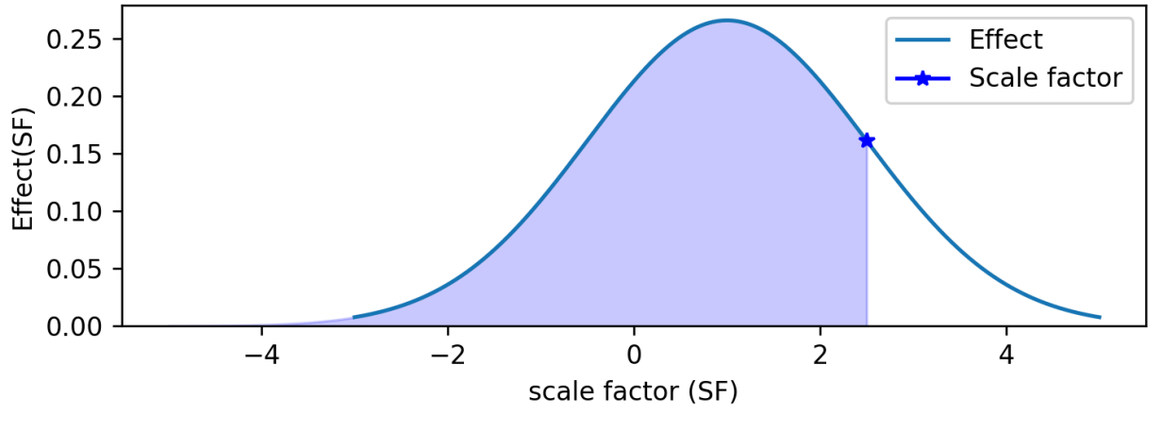
- θ : parameter, π: effect pdf, X: aux. measurement
- Visual example:

Constraint for $\mathcal{L}$



Eq. 1

SF for bins



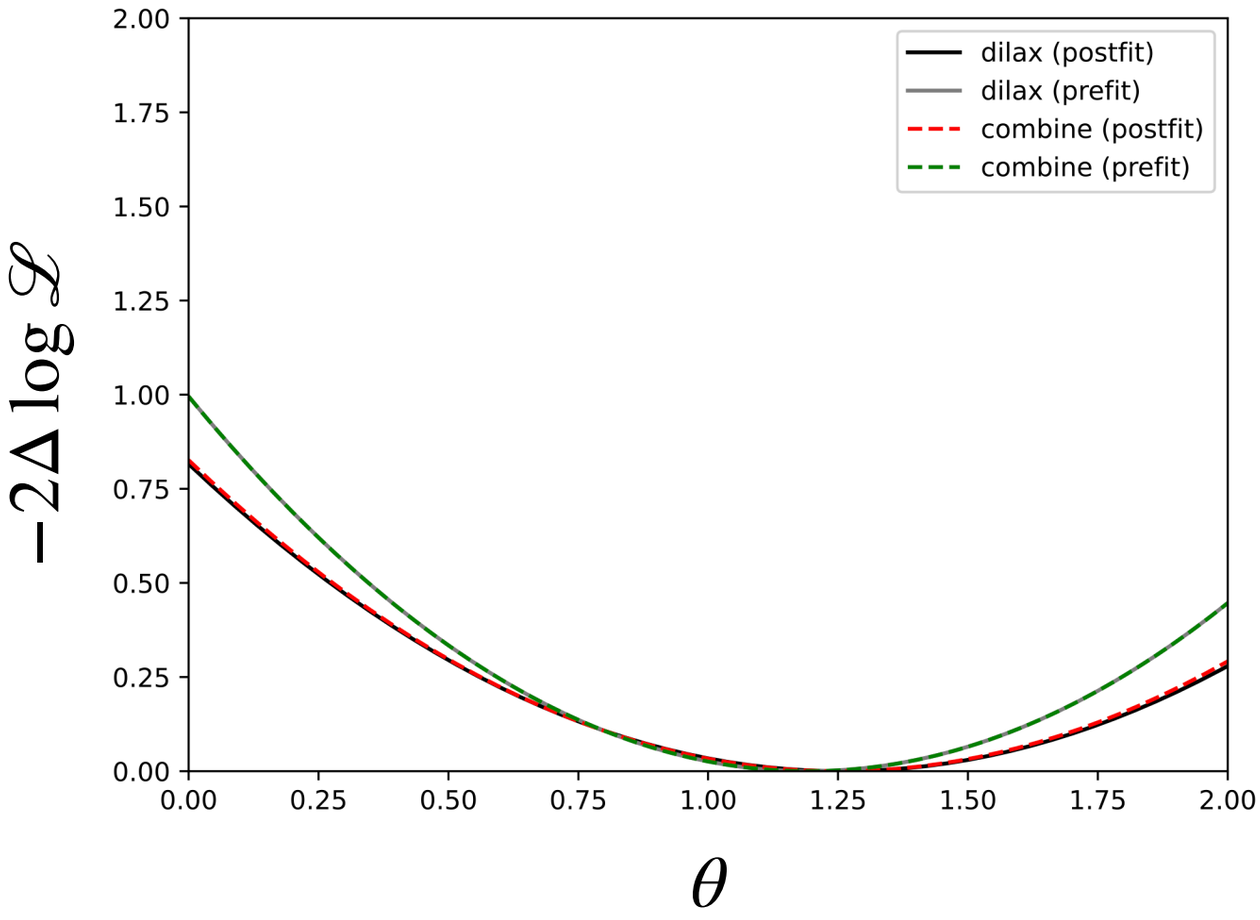
"A +1 sigma deviation of θ corresponds to a SF of 2.5 for a gaussian effect with width 1.5"

First Validation Checks (NLL profile)

Peter F. - 07.12.23

- Check likelihood profile of different parameter types (gauss, lnN, poisson, shape, ...) against CMS combine tool:

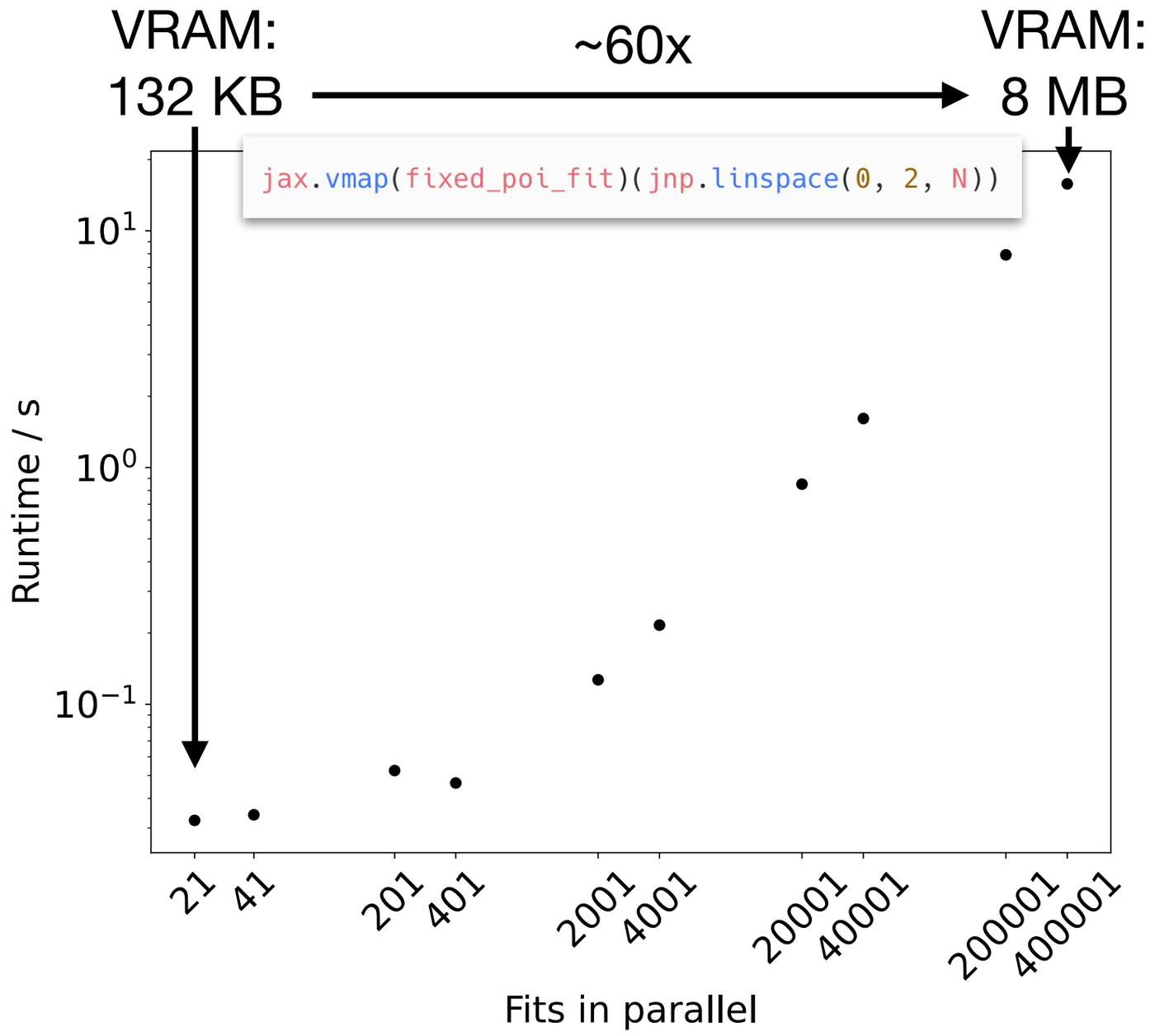
	Impl.	Val.
unconstrained	✓	✓
gauss	✓	✓
lnN	✓	✓
poisson	✓	✗
shape	✓	✗
Barlow-Beeston	✓	✗



- Currently setting up a benchmarking & testing suite for combine (and pyhf in the future)

Performance (NLL profile)

- Likelihood profile for a Model with:
  - 1 signal process (modified by μ)
  - 100 background processes (modified by 10% lnNs each)
  - Each process has 100 bins
- Compute idea:
  - Vectorize full fits on a GPU
  - Utilize jax.vmap
  - Minimizer: Gradientdescent
- Results:
  - Runtime:
    - 400k fits in 10s
    - O(ms) - O(s)
  - Compiletime: ~20s
  - Small VRAM footprint
  - XLA (GPU) memory opt.



Roadmap

Timeline (order/priority: top → bottom):	Comments
1. Validation checks of: – Individual components of *dilax* – Full analyses (e.g. with likelihoods from HEPData)	~4 Weeks
2. CLs / Limit calculation (toy-based & asymptotic)	~1-3 Days
3. Converter: – Combine datacard → *dilax.Model* – pyhf json → *dilax.Model* – HS3 → *dilax.Model*	~1-2 Weeks Different package(s)?
4. Minuit in JAX	Contribute to JAXopt?
5. Visualizations (Profiles, Pulls & Impacts, Limits, ...)	Different package?

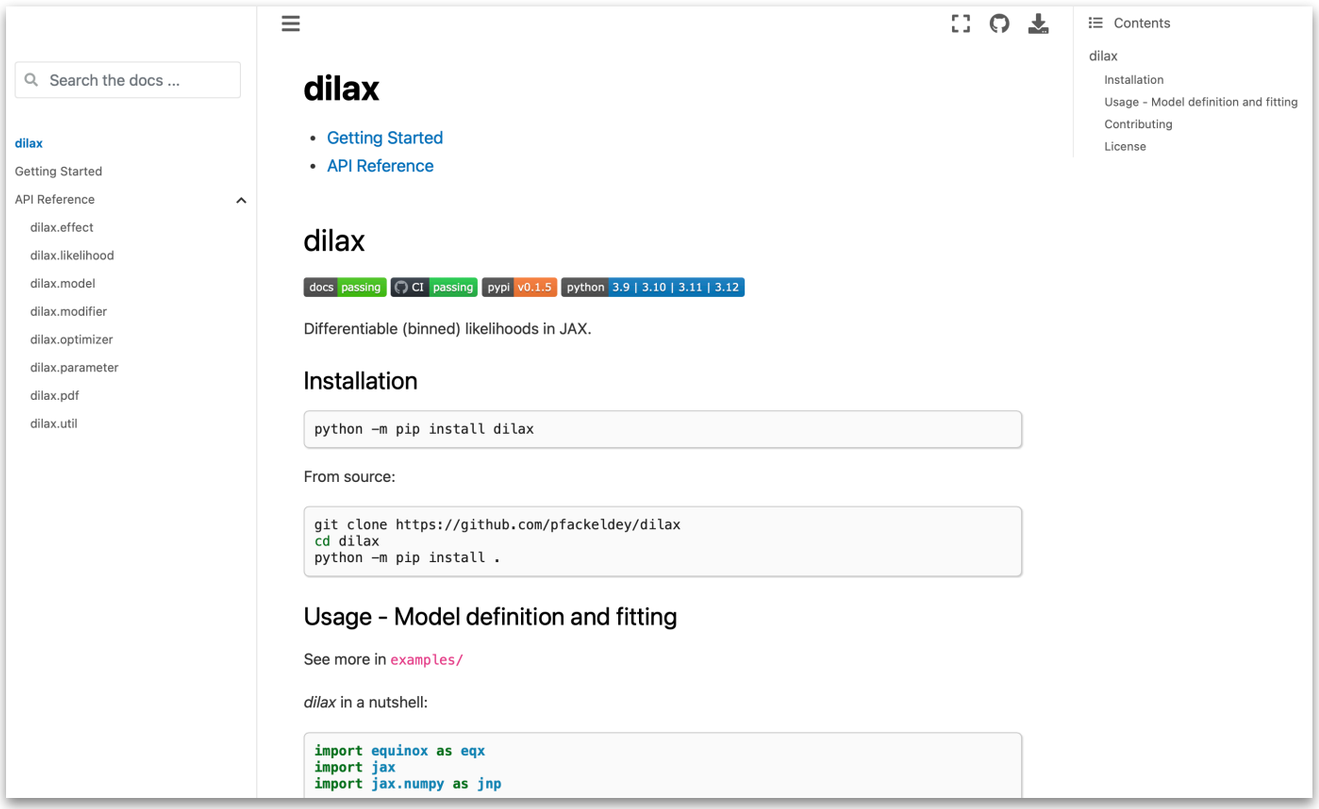
Current humanpower at RWTH Aachen University (Group: Prof. Erdmann):

- Me ('Post-PhD/Pre-Defense' phase)
- 1 new PhD student

Summary

- *dilax* is a fitting library for binned likelihood fits (in HEP)
- Purely based on JAX and the concept of PyTrees
- Key goals:
  - Performance (jax.jit, jax.vmap, ...)
  - Fully-differentiable (jax.grad, jax.hessian, ...)
  - Pythonic Model API (similar to torch.nn.Module)
  - Seamless integration into JAX-ecosystem

- Give it a try:
  - GitHub
  - Docs
  - Examples



Backup

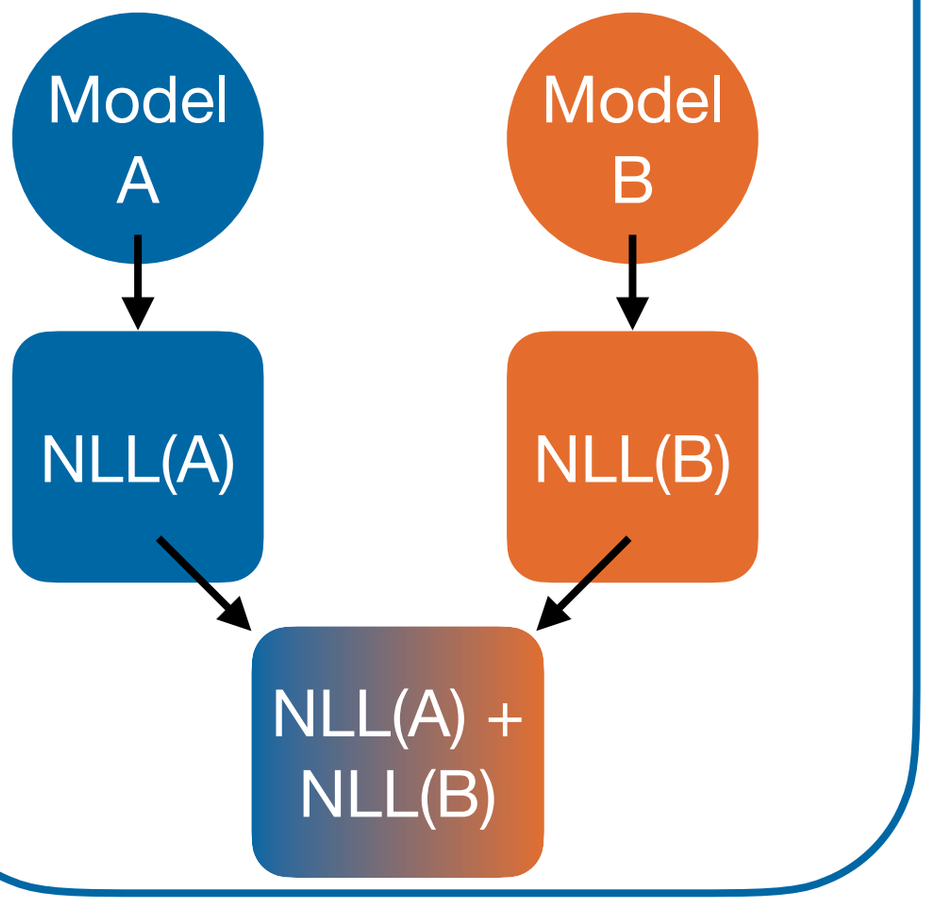
Combining Analyses

RWTH AACHEN UNIVERSITY
Peter F. - 07.12.23

Easier Implementation

Option 1:

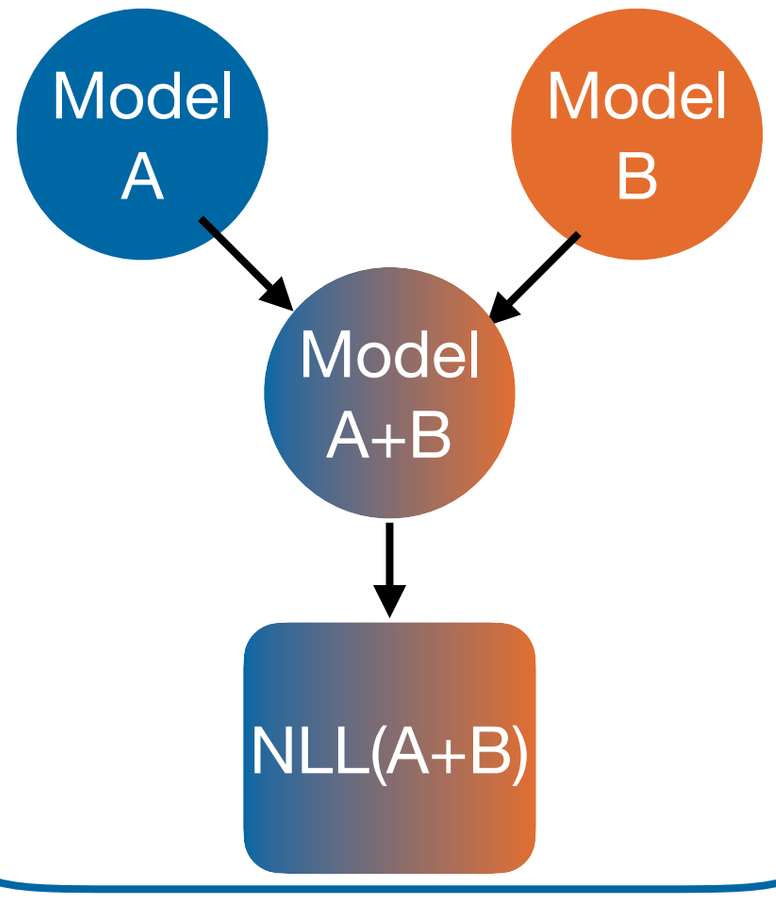
- One model per analyses/ channel/...
- Define NLL per model and sum all NLLs



Better Performance

Option 2:

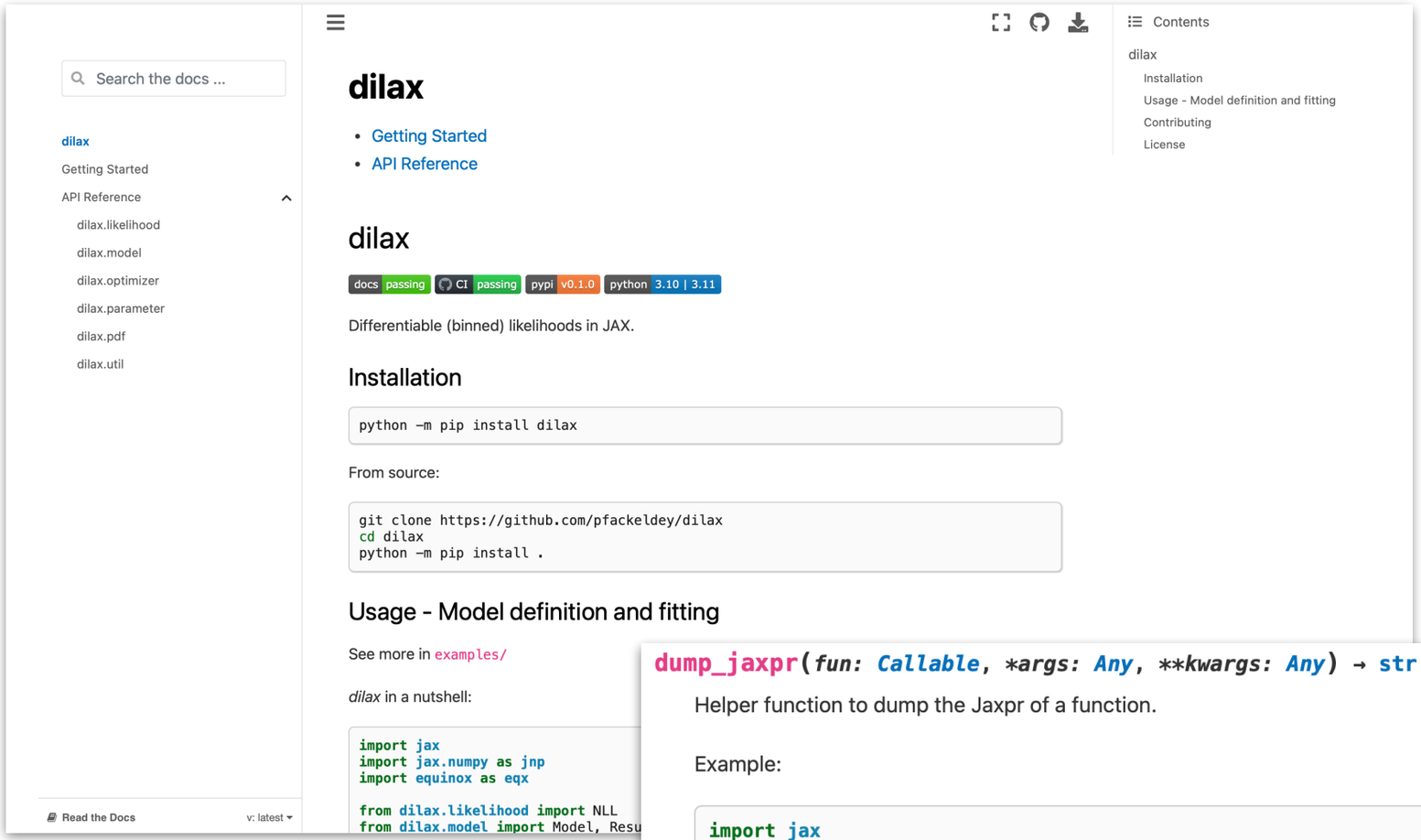
- One model for all analyses/ channel/...
- Vectorize along all bins

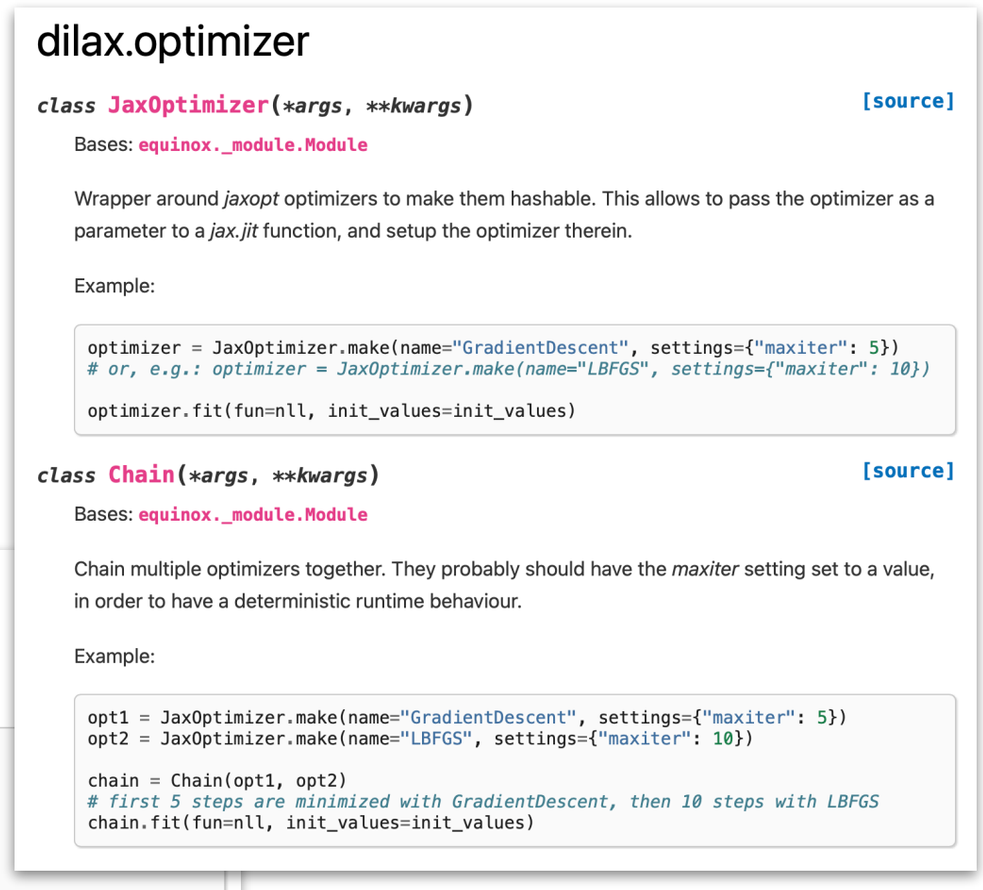


Documentation

RWTH AACHEN UNIVERSITY
Peter F. - 07.12.23

- dilax.readthedocs.io
- Hosted by *readthedocs*, made by *sphinx*
- Automatic API documentation, "Getting started" in development





Gradients of negative log-likelihood


```python
import equinox as eqx
import jax
import jax.numpy as jnp

import dilax as dlx

... # see previous slide

# fit
fitted_params, state = optimizer.fit(fun=fast_nll, init_values=init_values)


# gradients of "prefit" model:
grad_prefit_nll = eqx.filter_grad(fast_nll)


# gradients of "postfit" model:
postfit_model = model.update(values=fitted_params)

@eqx.filter_grad
@eqx.filter_jit
def grad_postfit_nll(where: dict[str, jax.Array]) -> dict[str, jax.Array]:
    nll = dlx.likelihood.NLL(model=postfit_model, observation=jnp.array([...]))
    return nll(where)
```

Likelihood profiling (vectorized)


```python
def nll_profiling(
    value_name: str,
    scan_points: jax.Array,
    model: Model,
    observation: jax.Array,
    optimizer: JaxOptimizer,
) -> jax.Array:

    # define single fit for a fixed parameter of interest (poi)
    @partial(jax.jit, static_argnames=("value_name", "optimizer"))
    def fixed_poi_fit(
        value_name: str,
        scan_point: jax.Array,
        model: Model,
        observation: jax.Array,
        optimizer: JaxOptimizer,
    ) -> jax.Array:
        ...

    # vectorise for multiple fixed values (scan points)
    fixed_poi_fit_vec = jax.vmap(
        fixed_poi_fit, in_axes=(None, 0, None, None, None)
    )
    return fixed_poi_fit_vec(
        value_name, scan_points, model, observation, optimizer
    )
```

dilax Model example


```python
# define a simple model with two processes and two parameters
class MyModel(dlx.Model):
    def __call__(
        self, processes: dict, parameters: dict[str, dlx.Parameter]
    ) -> dlx.Result:
        res = dlx.Result()

        # signal
        mu_mod = dlx.modifier(
            name="mu", parameter=parameters["mu"], effect=dlx.effect.unconstrained()
        )
        res.add(process="signal", expectation=mu_mod(processes["signal"]))

        # background
        bkg_mod = dlx.modifier(
            name="sigma", parameter=parameters["sigma"], effect=dlx.effect.gauss(0.2)
        )
        res.add(process="background", expectation=bkg_mod(processes["background"]))
        return res


# setup model
processes = {"signal": jnp.array([10.0]), "background": jnp.array([50.0])}
parameters = {
    "mu": dlx.Parameter(value=jnp.array([1.0]), bounds=(0.0, jnp.inf)),
    "sigma": dlx.Parameter(value=jnp.array([0.0])),
}
model = MyModel(processes=processes, parameters=parameters)
```

Nuisance parameters

- Correlations are introduced by reusing a parameter with a different effect

```python
norm = dlx.Parameter(value=0.0, bounds=(-jnp.inf, jnp.inf))

# lnN effect for QCD
modify_QCD = dlx.modifier(name="norm", parameter=norm, effect=dlx.effect.lnN(0.2))

# shape effect for DY
DY_up = jnp.array([12, 23, 35])
DY_down = jnp.array([8, 19, 26])
modify_DY = dlx.modifier(name="norm", parameter=norm, effect=dlx.effect.shape(DY_up, DY_down))
```

- Multiple modifiers can be composed into one modifier

```python
mu = dlx.Parameter(value=1.1, bounds=(0, 100))
sigma = dlx.Parameter(value=0.1, bounds=(-100, 100))

# create a new parameter and a composition of modifiers
composition = dlx.compose(
    dlx.modifier(name="mu", parameter=mu),
    dlx.modifier(name="sigma1", parameter=sigma, effect=dlx.effect.lnN(0.1)),
)

# nest compositions
composition = dlx.compose(
    composition,
    dlx.modifier(name="sigma2", parameter=sigma, effect=dlx.effect.lnN(0.2)),
)

# apply the composition
composition(jnp.array([10, 20, 30]))
```

Interactive *dilax.Model* (IPython)

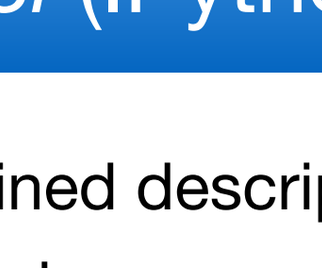
- A *dilax.Model* is a self-contained description of all histograms (S+B model) including all nuisance parameters

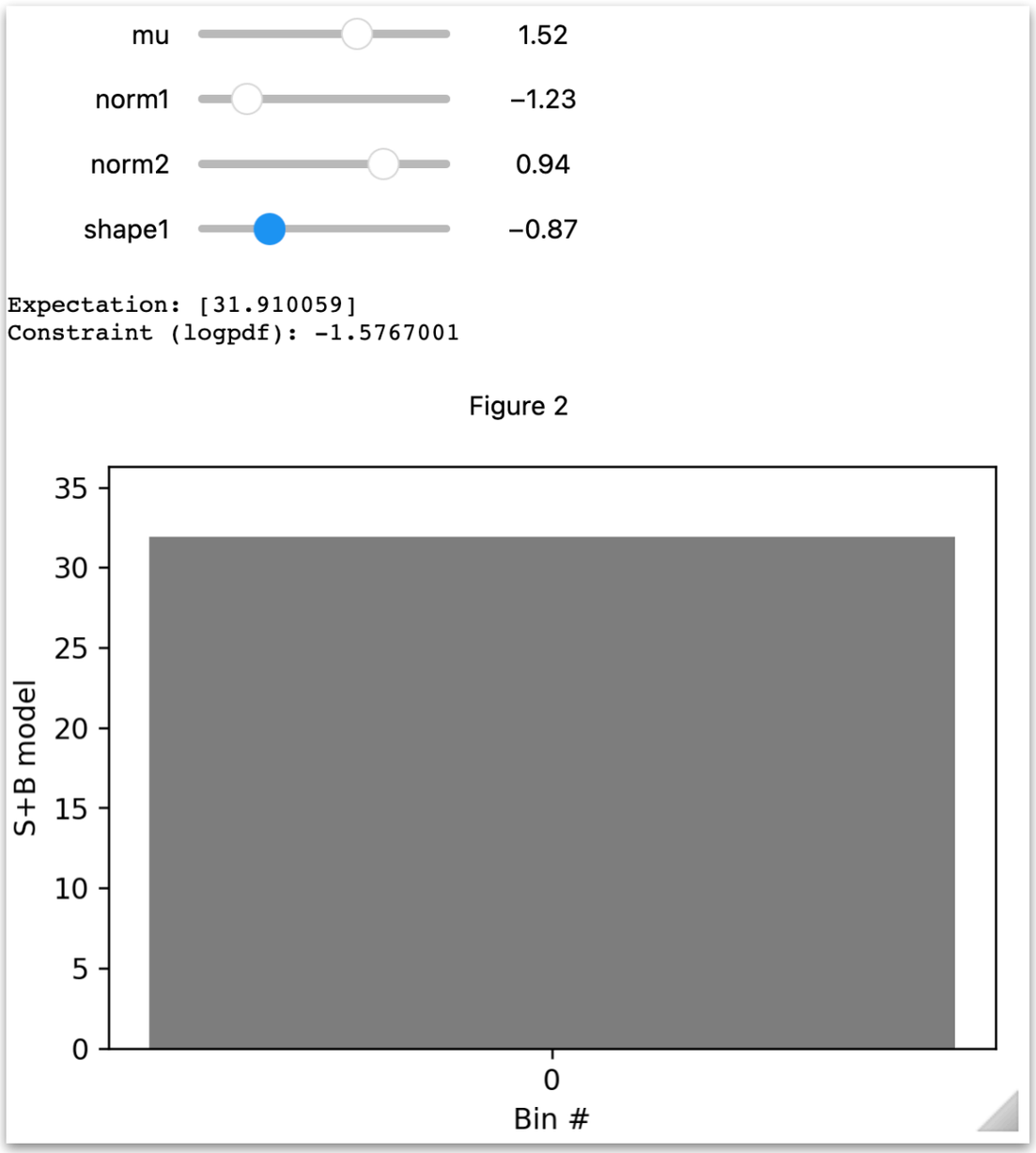
  → add interactivity for IPython/Jupyter notebooks with a single function:

```
%matplotlib widget

from model import model
from dilax.ipy_util import interactive

interactive(model)
```





(*Still needs proper plotting...*)

MC statistics uncertainties

- Automatically calculable by *dilax* (needs sumw & sumw2 of each process)
- 3 possible modes (dlx.autostaterrors.Mode):
  1. poisson: Use poisson effect for every bin and process
  2. poisson_gauss: Use poisson (gaussian) effect for every bin and process if bin content < (>=) threshold
  3. barlow_beeston_lite: Barlow-beeston-lite approach

```
sumw = {
    "signal": jnp.array([5, 20, 30]),
    "background": jnp.array([5, 20, 30]),
}

sumw2 = {
    "signal": jnp.array([5, 20, 30]),
    "background": jnp.array([5, 20, 30]),
}


auto = dlx.autostaterrors(
    sumw=sumw,
    sumw2=sumw2,
    threshold=10.0,
    mode=dlx.autostaterrors.Mode.poisson,
)
parameters, staterror_modifiers = auto.prepare()
```

Mode