~~dilax~~ evermore

Peter Fackeldey, Benjamin Fischer,
Felix Zinn, Martin Erdmann

ACAT 2024

14.03.24

RWTH AACHEN UNIVERSITY

evermore at a glance

What is evermore?

- Python library (*pip install evermore*)
- Provides tools for statistical model & likelihood building
- Based on JAX (numpy/scipy with JIT & autodiff.) and Equinox
- Integrates nicely with JAX ecosystem, e.g.:
  - Optimizers: JAXopt, Optimistix, Optax
  - Utilities: chex, orbax

Key Concept:

- *Everything* in evermore is a JAX PyTree ("Models as PyTrees" → backup)

  → fully compatible with JAX composable transformations
  - Differentiability (jax.grad): gradients through likelihoods (like *neos*)
  - Performance (jax.jit, jax.vmap): GPU acceleration, vectorized fitting, ...

HEP Fitting Ingredients

JAX

Equinox

evermore

Model definition

Likelihood definition

iminuit

JAXopt

Optax

Optimistix

Minimization

Model Definition

Models as PyTrees: evermore Example

- Represent statistical models as PyTrees (recommendation: eqx.Module)
- Leaves are (primarily) nuisance parameters → evm.Parameter
- Example: PyTree with leaves mu and syst

```
class Model(eqx.Module):
    mu: evm.Parameter
    syst: evm.Parameter
```

Step 1: Define PyTree with leafs

Models as PyTrees: evermore Example

- Represent statistical models as PyTrees (recommendation: eqx.Module)
- Leaves are (primarily) nuisance parameters → evm.Parameter
- Example: PyTree with leaves mu and syst

```python
class Model(eqx.Module):
    mu: evm.Parameter
    syst: evm.Parameter

    def __call__(self, hists: dict[str, Array]) -> Array:
        mu_modifier = self.mu.unconstrained()
        syst_modifier = self.syst.lnN(width=jnp.array([0.9, 1.1]))
        return mu_modifier(hists["signal"]) + syst_modifier(hists["bkg"])
```

Step 2: Define how S+B is calculated

Models as PyTrees: evermore Example

- Represent statistical models as PyTrees (recommendation: eqx.Module)
- Leaves are (primarily) nuisance parameters → evm.Parameter
- Example: PyTree with leaves mu and syst

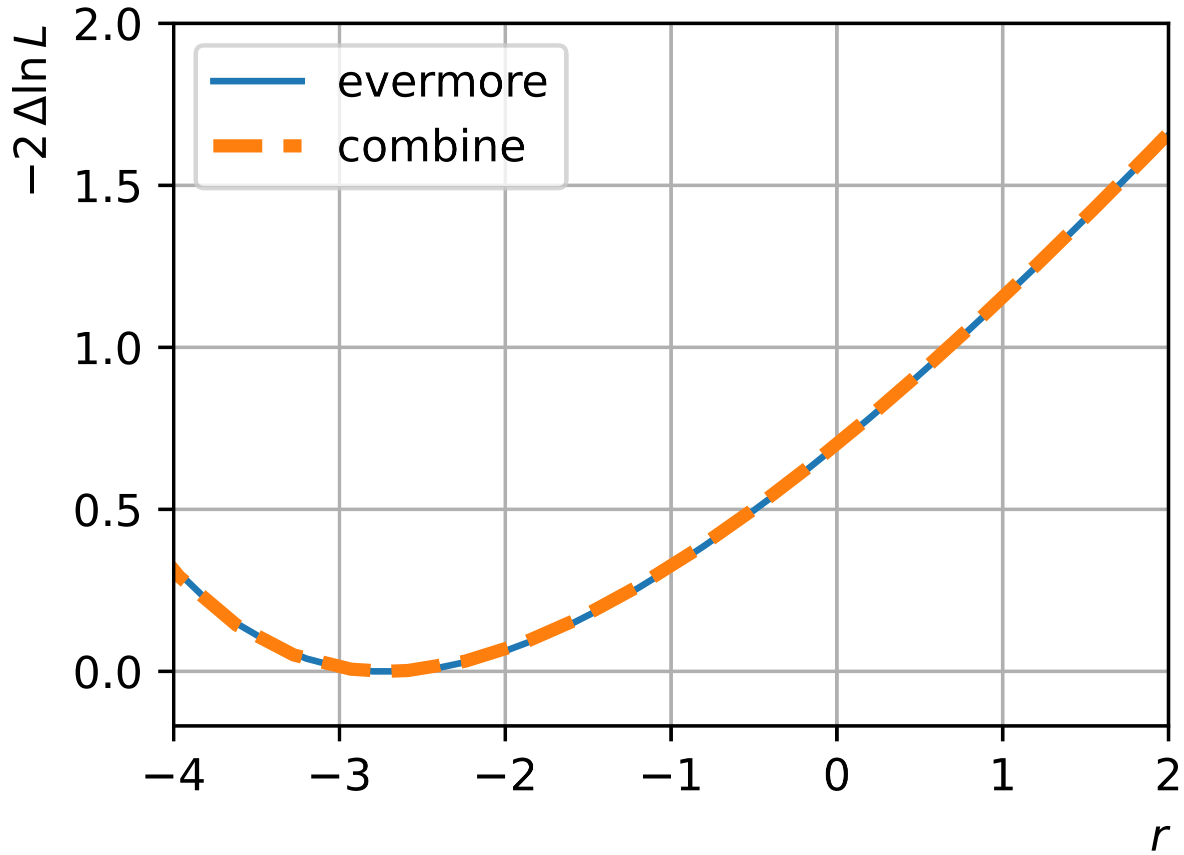
```python
class Model(eqx.Module):
    mu: evm.Parameter
    syst: evm.Parameter

    def __call__(self, hists: dict[str, Array]) -> Array:
        mu_modifier = self.mu.unconstrained()
        syst_modifier = self.syst.lnN(width=jnp.array([0.9, 1.1]))
        return mu_modifier(hists["signal"]) + syst_modifier(hists["bkg"])


model = Model(mu=evm.Parameter(1.0), syst=evm.Parameter(0.0))
hists = {"signal": jnp.array([3]), "bkg": jnp.array([10])}
print(f"Expectation: {model(hists)}")
# -> Expectation: [13.]
```

Step 3: Evaluate model with histograms

Modifiers: Basics

- evm.modifier defines how bins are modified
- evm.modifier use a evm.Parameter and a evm.effect (e.g. gauss or lnN)
- Correlated evm.modifier(s): use the same evm.Parameter with a different evm.effect

- Available evm.effect(s): <u>Validated</u>
  - unconstrained ✓
  - gauss ✓
  - lnN ✓
  - shape ✓
  - poisson ✗



- Validated against CMS combine tool

Modifiers: Advanced Concepts

- Advanced modifier concepts/transformations:
  - Combine modifiers into new modifier: evm.modifier.compose
  - Apply modifiers based on condition: evm.modifier.where
  - Apply a modifier only in certain bins: evm.modifier.mask
  - Transform modifier based on function: evm.modifier.transform
- Barlow-Beeston-Lite implementation with evm.modifier.where

```python
syst = evm.Parameter()
lnN_mod1 = syst.lnN(width=jnp.array([0.9, 1.1]))
lnN_mod2 = syst.lnN(width=jnp.array([0.8, 1.2]))

hist = jnp.array([3, 12, 50])
lnN_composition = evm.modifier.compose(lnN_mod1, lnN_mod2)  # `compose`
lnN_where = evm.modifier.where(hist > 10, lnN_mod1, lnN_mod2)  # `where`
lnN1_mask = evm.modifier.mask(jnp.array([True, False, True]), lnN_mod1)  # `mask`
lnN1_sqrt_mod = evm.modifier.transform(jnp.sqrt, lnN_mod1)  # `transform`
# `clip`
clip = partial(jnp.clip, a_min=0.8, a_max=1.2)
lnN1_clipped = evm.modifier.transform(clip, lnN_mod1)
```

Likelihood Definition ("Loss function")

Negative Log-Likelihood ("loss")

$$\log \mathscr{L} = \sum \text{Poisson}(d_i, \lambda_i(s_i, b_i, \vec{\theta}))$$

```python
nll = evm.loss.PoissonNLL()


def loss(model: Model, hists: dict[str, Array], observation: Array) -> Array:
    expectation = model(hists)
    # Poisson NLL of the expectation and observation
    log_likelihood = nll(expectation, observation)
```

Step 1: Define Poisson negative log-likelihood

Negative Log-Likelihood ("loss")

$$\log \mathscr{L} = \sum \text{Poisson}(d_i, \lambda_i(s_i, b_i, \vec{\theta})) + \sum_j \pi_j(\theta_j)$$

```
nll = evm.loss.PoissonNLL()


def loss(model: Model, hists: dict[str, Array], observation: Array) -> Array:
    expectation = model(hists)
    # Poisson NLL of the expectation and observation
    log_likelihood = nll(expectation, observation)
    # Add parameter constraints from logpdfs
    constraints = evm.loss.get_param_constraints(model)
    log_likelihood += evm.util.sum_leaves(constraints)
    return -jnp.sum(log_likelihood)
```

Step 2: Add parameter constraints and sum up

Putting the pieces together

Model definition

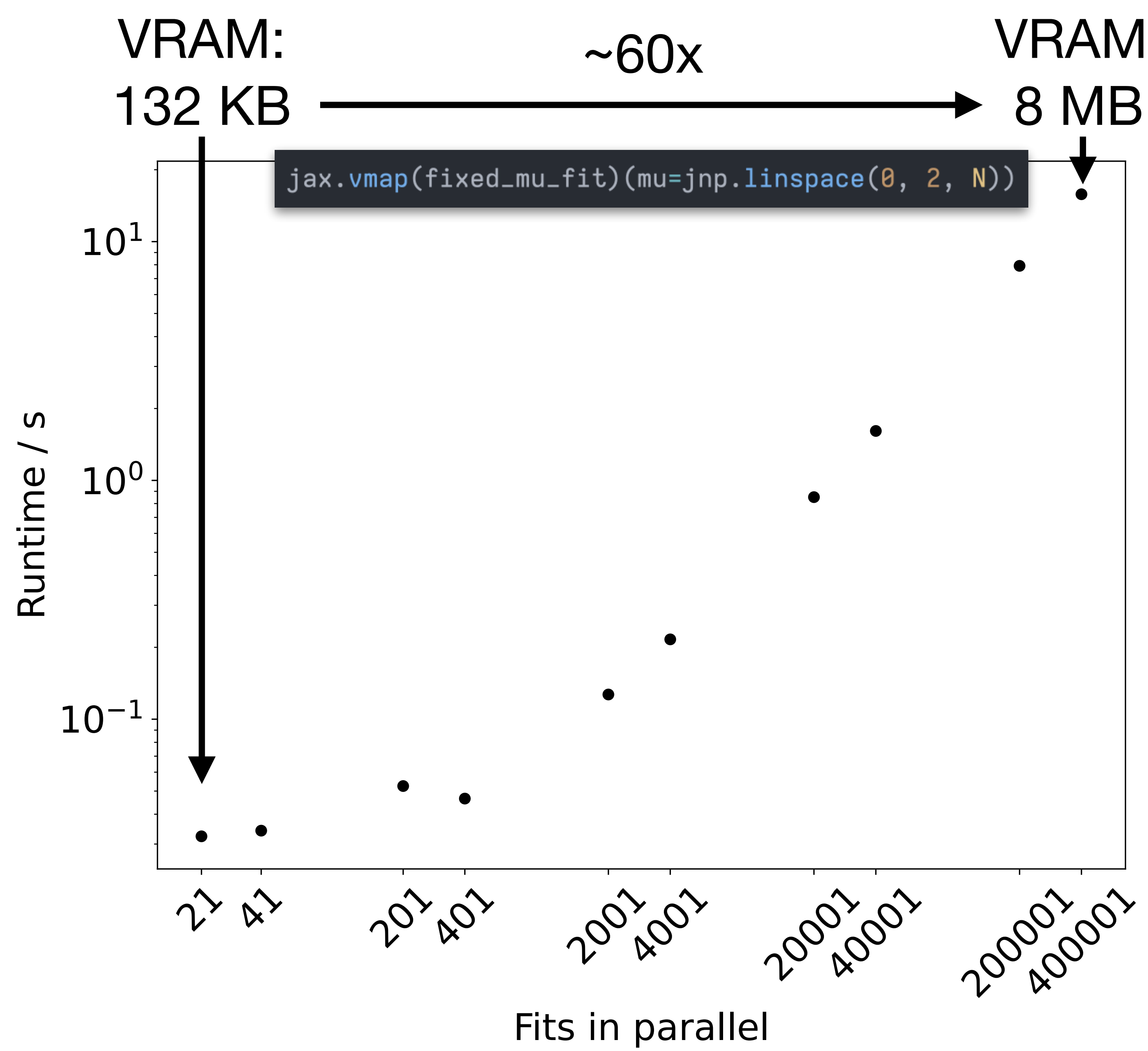
```python
class Model(eqx.Module):
    mu: evm.Parameter
    syst: evm.Parameter

    def __call__(self, hists: dict[str, Array]) -> Array:
        mu_modifier = self.mu.unconstrained()
        syst_modifier = self.syst.lnN(width=jnp.array([0.9, 1.1]))
        return mu_modifier(hists["signal"]) + syst_modifier(hists["bkg"])


nll = evm.loss.PoissonNLL()
```

Likelihood definition

```python
def loss(model: Model, hists: dict[str, Array], observation: Array) -> Array:
    expectation = model(hists)
    # Poisson NLL of the expectation and observation
    log_likelihood = nll(expectation, observation)
    # Add parameter constraints from logpdfs
    constraints = evm.loss.get_param_constraints(model)
    log_likelihood += evm.util.sum_leaves(constraints)
    return -jnp.sum(log_likelihood)
```

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.



Constrained DNN weights


```python
class LinearConstrained(eqx.Module):
    weights: evm.Parameter
    biases: jax.Array

    def __init__(self, in_size, out_size, key):
        wkey, bkey = jax.random.split(key)
        # weights
        constraint = evm.pdf.Gauss(
            mean=jnp.zeros((out_size, in_size)),
            width=jnp.full((out_size, in_size), 0.5),
        )
        self.weights = evm.Parameter(
            value=jax.random.normal(wkey, (out_size, in_size)), constraint=constraint
        )

        # biases
        self.biases = jax.random.normal(bkey, (out_size,))

    def __call__(self, x: jax.Array):
        return self.weights.value @ x + self.biases
```

Summary

- Evermore is a library for binned likelihood fits (in HEP)
- Based on JAX & Equinox, 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) with eqx.Module
  - Seamless integration into JAX-ecosystem

- Give it a try → *pip install evermore*
  - GitHub
  - Docs
  - Examples

Soon: *stress-test with large fit model (real analysis)*

Backup

Putting the pieces together: full example


```python
class Model(eqx.Module):
    mu: evm.Parameter
    syst: evm.Parameter

    def __call__(self, hists: dict[str, Array]) -> Array:
        mu_modifier = self.mu.unconstrained()
        syst_modifier = self.syst.lnN(width=jnp.array([0.9, 1.1]))
        return mu_modifier(hists["signal"]) + syst_modifier(hists["bkg"])


nll = evm.loss.PoissonNLL()


def loss(model: Model, hists: dict[str, Array], observation: Array) -> Array:
    expectation = model(hists)
    # Poisson NLL of the expectation and observation
    log_likelihood = nll(expectation, observation)
    # Add parameter constraints from logpdfs
    constraints = evm.loss.get_param_constraints(model)
    log_likelihood += evm.util.sum_leaves(constraints)
    return -jnp.sum(log_likelihood)


@eqx.filter_jit
def make_step(
    model: Model, opt_state: PyTree, hists: dict[str, Array], observation: Array
) -> PyTree:
    # differentiate full model
    grads = eqx.filter_grad(loss)(model, hists, observation)
    updates, opt_state = optim.update(grads, opt_state)
    # apply nuisance parameter updates
    model = eqx.apply_updates(model, updates)
    return model, opt_state


model = Model(mu=evm.Parameter(1.0), syst=evm.Parameter(0.0))
hists = {"signal": jnp.array([3]), "bkg": jnp.array([10])}
observation = jnp.array([15])

optim = optax.sgd(learning_rate=1e-2)
opt_state = optim.init(eqx.filter(model, eqx.is_inexact_array))

# minimize model with 100 steps
for step in range(100):
    model, opt_state = make_step(model, opt_state, hists, observation)
print(f"{model.mu.value=}, {model.syst.value=}")
# -> model.mu.value=[1.316672], model.syst.value=[0.06218078]
```

Model definition

Likelihood definition

Minimization

More Complete Model Example


```python
class SPlusBModel(eqx.Module):
    mu: evm.Parameter
    norm1: evm.Parameter
    norm2: evm.Parameter
    shape1: evm.Parameter

    def __init__(self, hist: dict[str, Array], histw2: dict[str, Array]) -> None:
        self.mu = evm.Parameter(value=jnp.array([1.0]))
        self = evm.parameter.auto_init(self)

    def __call__(self, hists: dict) -> dict[str, Array]:
        expectations = {}

        # signal process
        sig_mod = self.mu.unconstrained()
        expectations["signal"] = sig_mod(hists["nominal"]["signal"])

        # bkg1 process
        bkg1_lnN = self.norm1.lnN(width=jnp.array([0.9, 1.1]))
        bkg1_shape = self.shape1.shape(
            up=hists["shape_up"]["bkg1"],
            down=hists["shape_down"]["bkg1"],
        )
        # combine modifiers
        bkg1_mod = bkg1_lnN @ bkg1_shape
        expectations["bkg1"] = bkg1_mod(hists["nominal"]["bkg1"])

        # bkg2 process
        bkg2_lnN = self.norm2.lnN(width=jnp.array([0.95, 1.05]))
        bkg2_shape = self.shape1.shape(
            up=hists["shape_up"]["bkg2"],
            down=hists["shape_down"]["bkg2"],
        )
        # combine modifiers
        bkg2_mod = bkg2_lnN @ bkg2_shape
        expectations["bkg2"] = bkg2_mod(hists["nominal"]["bkg2"])

        # return the modified expectations
        return expectations
```

Models as PyTrees: What are PyTrees?

- PyTrees are "tree-like structures built out of container-like Python objects"
- Many JAX functions operate on PyTrees of jax.Array(s)
- Many PyTree manipulation tools (jax.tree_util, eqx.{partition,combine,filter})
- Custom PyTrees by providing tree_flatten & tree_unflatten methods

What I (user) see

```python
pytree = {
    "foo": jnp.array([1, 2, 3]),
    "bar": {
        "a": jnp.array([4, 5, 6]),
        "b": jnp.array([7, 8, 9]),
    },
}


def fun(pytree: PyTree) -> Array:
    x = pytree["bar"]["a"] + pytree["bar"]["b"]
    return x * pytree["foo"]

print(jax.make_jaxpr(fun)(pytree))
```

What JAX does

- "pytree" gets flattened
- Only leafs enter function
- Transforms fun:

  1 arg (PyTree) → 3 args (Array)

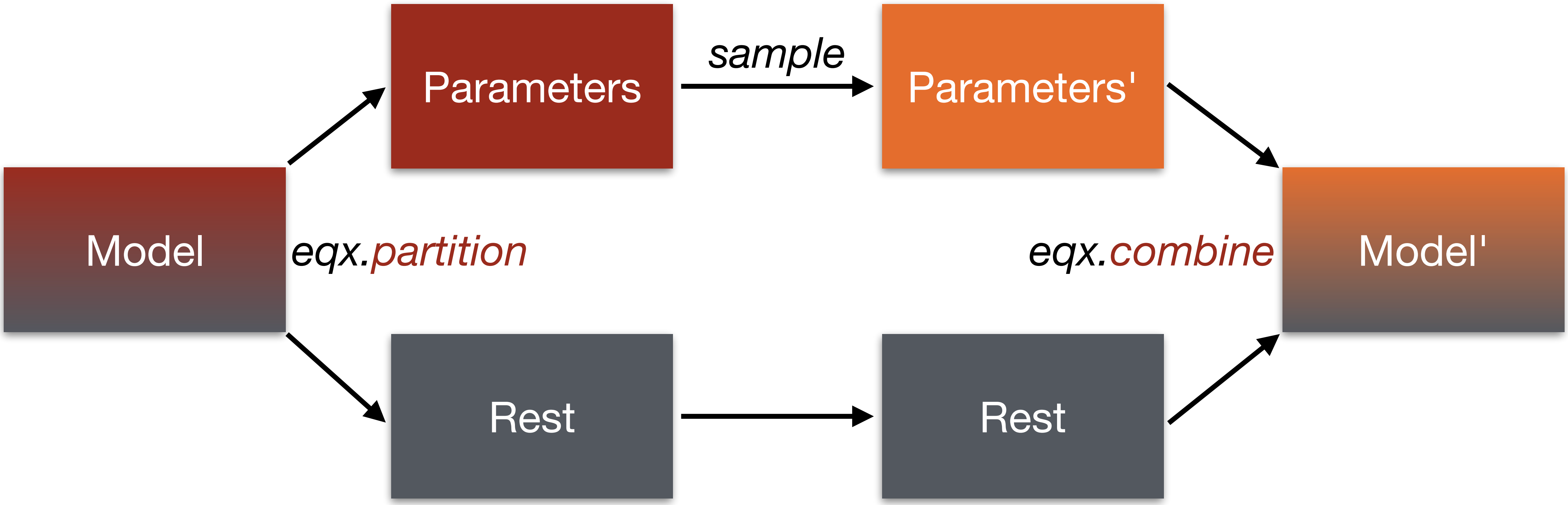
translated

```
{ lambda ; a:i32[3] b:i32[3] c:i32[3]. let
    d:i32[3] = add a b
    e:i32[3] = mul d c
  in (e,) }
```

Models as PyTrees: Benefits

- Compatible with JAX transformations (jax.jit, jax.grad, jax.vmap, ...)
- Model surgery, e.g.:
  - Interact only with the leafs of type evm.Parameter (extract constraints, sampling, ...)
  - Freeze evm.Parameter during minimisation
- A model can hold additional information, e.g., DNN weights ("Rest")

Example: evm.Parameter sampling



Partition PyTrees

model

```
{
    "foo": jnp.array([1, 2, 3]),
    "bar": evm.Parameter(),
    "baz": {
        "a": jnp.array([4, 5, 6]),
        "b": [evm.Parameter(), jnp.array([1.0])],
        "c": None,
    },
}
```

eqx.partition

parameter leafs

```
{
    "foo": None,
    "bar": evm.Parameter(),
    "baz": {
        "a": None,
        "b": [evm.Parameter(), None],
        "c": None,
    },
}
```

other leafs

```
{
    "foo": jnp.array([1, 2, 3]),
    "bar": None,
    "baz": {
        "a": jnp.array([4, 5, 6]),
        "b": [None, jnp.array([1.0])],
        "c": None,
    },
}
```

Combine PyTrees

parameter leafs

```
{
    "foo": None,
    "bar": evm.Parameter(),
    "baz": {
        "a": None,
        "b": [evm.Parameter(), None],
        "c": None,
    },
}
```

other leafs

```
{
    "foo": jnp.array([1, 2, 3]),
    "bar": None,
    "baz": {
        "a": jnp.array([4, 5, 6]),
        "b": [None, jnp.array([1.0])],
        "c": None,
    },
}
```

eqx.combine

model

```
{
    "foo": jnp.array([1, 2, 3]),
    "bar": evm.Parameter(),
    "baz": {
        "a": jnp.array([4, 5, 6]),
        "b": [evm.Parameter(), jnp.array([1.0])],
        "c": None,
    },
}
```

Tipps & Tricks

Reduce Compiletime: avoid python loops wherever possible

Do

```python
@jax.jit
def fit(steps: int = 1000) -> tuple[eqx.Module, tuple]:
    def fun(step, model_optstate):
        model, opt_state = model_optstate
        return make_step(model, opt_state, hists, observation)

    return jax.lax.fori_loop(0, steps, fun, (model, opt_state))
```

Don't

```python
for step in range(1000):
    model, opt_state = make_step(model, opt_state, hists, observation)
```

Auto setup evm.Parameter: evm.parameter.auto_init(self)

Do

```python
class SPlusBModel(eqx.Module):
    mu: evm.Parameter
    norm1: evm.Parameter
    norm2: evm.Parameter
    shape1: evm.Parameter

    def __init__(self, hist: dict[str, Array], histw2: dict[str, Array]) -> None:
        self.mu = evm.Parameter(value=jnp.array([1.0]))
        self = evm.parameter.auto_init(self)
```

Don't

```python
class SPlusBModel(eqx.Module):
    mu: evm.Parameter
    norm1: evm.Parameter
    norm2: evm.Parameter
    shape1: evm.Parameter

    def __init__(self, hist: dict[str, Array], histw2: dict[str, Array]) -> None:
        self.mu = evm.Parameter(value=jnp.array([1.0]))
        self.norm1 = evm.Parameter()
        self.norm2 = evm.Parameter()
        self.shape1 = evm.Parameter()
```

Other tricks: https://docs.kidger.site/equinox/tricks/

Fully factorized constraint concept

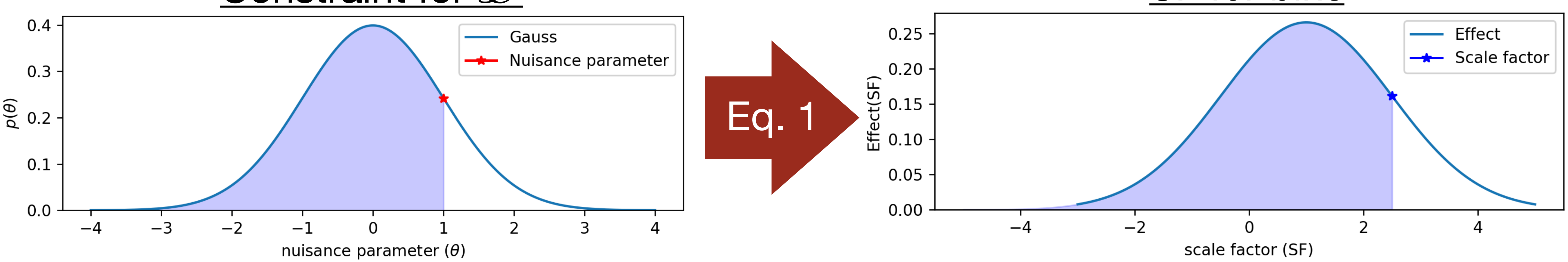
RWTH AACHEN UNIVERSITY
Peter F. - 14.03.24

- 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:

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

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

<u>Constraint for $\mathscr{L}$</u> <u>SF for bins</u>



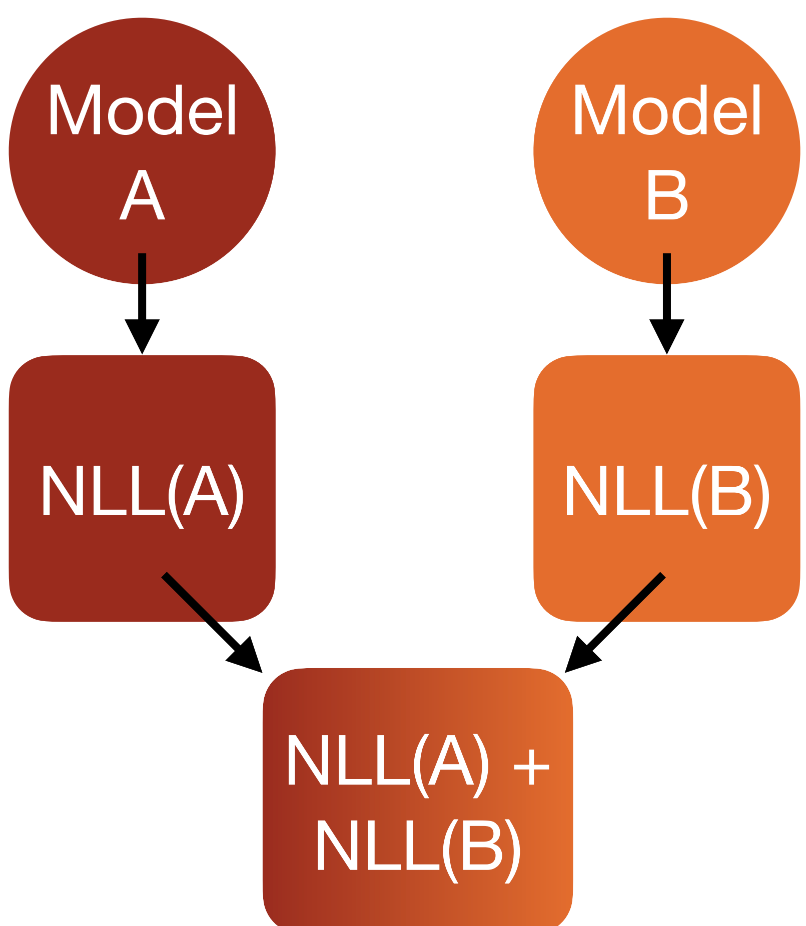
"A +1 sigma deviation of θ corresponds to a SF of 2.5 for a gaussian effect with width 1.5"

Combining Analyses

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