FlatPPL for Python

SCALAR QUERY

$$\ell(x, \mu) = -\frac{(x-\mu)^2 + \log(2\pi)}{2}$$

COMPLETE EXAMPLE

A FlatPPL query as a JAX function

```
1  import jax
2  import jax.numpy as jnp
3  from flatppl import flatppl
5  model = flatppl(r"""
6  flatppl_compat = "0.1"
7  x = elementof(reals)
8  mu = elementof(reals)
9  inputs = (x, mu)
10 outputs = logdensityof(Normal(mu, 1.0), x)
11 """)
A 12 score = model.compile()
B 14 value = score(x=1.0, mu=0.0)
15 print(float(value))
C 17 gradient = jax.value_and_grad(score, argnums=0)
18 value, dx = jax.jit(gradient)(1.0, 0.0)
19 print(float(dx))
D 21 xs = jnp.array([-1.0, 0.0, 1.0])
22 batch = jax.vmap(score, in_axes=(0, None))
23 print(batch(xs, 0.0))
```

The FlatPPL interface

```
inputs = (x, mu)
outputs = logdensityof(Normal(mu, 1.0), x)
```

The tuple fixes the positional argument order. Each call supplies both values to the same compiled query.

Batch values and gradients together

```
both = jax.vmap(gradient, in_axes=(0, None))
values, slopes = both(xs, 0.0)
```

This returns two arrays of shape `(3,)`. Read the scores and slopes in the same order as `xs`.

Save and reload a model

```
from flatppl import load
model.save("normal.flatppl.zip")
f = load("normal.flatppl.zip").compile()
```

`save` writes a portable ZIP with the model and its dependencies. `load` opens a ZIP or a plain `.flatppl` file. The restored query accepts the same inputs.

Run the driver

```
python python_sdk.py
```

Save the left listing as `python_sdk.py`, or run it in a notebook. The SDK installation guide lists the required packages.

ANNOTATED LINES

A Compile the declared query — 12

```
score = model.compile()
```

Pass a raw string to `flatppl`. Call `compile()` to get a function with the declared `inputs`. Each call evaluates the `outputs` expression.

B Pass a point by name — 14–15

```
value = score(x=1.0, mu=0.0)
print(float(value))
```

Keyword arguments use the FlatPPL input names. The equivalent positional call is `score(1.0, 0.0)`. The result is a scalar JAX array. `float` converts it for printing.

C Differentiate one argument — 17–19

```
gradient = jax.value_and_grad(score, argnums=0)
value, dx = jax.jit(gradient)(1.0, 0.0)
print(float(dx))
```

`argnums=0` selects the first positional input, `x`. The transformed call returns the log density and its derivative. At this point the derivative is −1.

D Batch over the observations — 21–23

```
xs = jnp.array([-1.0, 0.0, 1.0])
batch = jax.vmap(score, in_axes=(0, None))
print(batch(xs, 0.0))
```

`in_axes=(0, None)` takes successive values of `x` from the first axis of `xs` and reuses one scalar mean. Three inputs give three log densities.

Inspect a binding

```
names = tuple(model.parameters)
domain = model.bindings["x"].domain
```

Here `names` is `("x", "mu")`. The binding for `x` records its real domain. Read the FlatPPL text from `model.source`.

Construct a regression from columns

```
from flatppl import from_formula
data = {"x": [0., 1., 2.], "y": [1., 2., 4.]}
reg = from_formula("y ~ x", data)
```

The default family is Gaussian. The formula includes an intercept and one slope. Inspect the generated model in `reg.source`.

RESULTS AND API CALLS

Check the values at μ = 0

x	Log density	Slope in x
-1	−1.41894	$+1$
0	−0.91894	0
1	−1.41894	-1

The batched call returns the middle column. The derivative is $\mu - x$, so opposite displacements have equal scores and opposite slopes.

Query a new expression

```
q = model.query({"hi": "mu + 2.0"})
hi = q.compile()({"mu": 1.0})["hi"]
```

The SDK builds a query over the parameters the expression uses. This one takes a record with one field and returns 3.

Bind an observed reading

```
m = flatppl(r"""
mu = elementof(reals)
datum = external(reals)
K = functionof(Normal(mu, 1.0))
L = likelihoodof(K, datum)
prior = joint(mu = Normal(0.0, 1.0))
posterior = bayesupdate(L, prior)
""").set(datum=1.0)
```

`set` binds one observed value to the external `datum`. The mean remains free.

Fit or sample the mean

```
target = m.logdensity("L", jacobian=False)
post = m.logdensity("posterior")
```

`logdensity` builds an inference target from a named law or likelihood. Minimize `target.potential` for the fit. Start a sampler at `post.init` and map draws with `post.constrain`. The posterior has mean and variance $\frac{1}{2}$.

Choose precision

```
with jax.enable_x64(True):
    f64 = model.compile(dtype="float64")
    value64 = f64(1.0, 0.0)
```

The default is float32. For array inputs, match the declared shape and the compiled dtype.

Jessica Cox · bcox@mpp.mpg.de