# AutoConj: find and exploit exponential family structure without a DSL

Matthew D. Hoffman*, Matthew J. Johnson*, Dustin Tran

Google AI · Google brain

**TL;DR** Write models in regular Python+Numpy with no mini-language, get exponential family structure-exploiting inference algorithms.

**Why?** Exploiting exponential family structure when it exists is labor-intensive, even for experts, which limits how we design new models and try new hybrid inference strategies (e.g. SVAEs). It's like neural nets before autodiff.

What is the autodiff for exponential family inference? AutoConj!

**DSL?** As with autodiff, don't want to be locked-in to a mini-language:

- New inference algorithms? Model classes?
- Optimization libraries? Automatic differentiation? Viz.?
- Compile to accelerators, distributed computing?

Need a system in native Python, and composable with others.

## Background: exponential families

Define a probability model via a **statistic function** $t(x)$

$$p(x;\eta) = \exp\left\{\langle \eta,\, t(x)\rangle - \mathcal{A}(\eta)\right\}, \qquad \mathcal{A}(\eta) \triangleq \log \int \exp\left\{\langle \eta,\, t(x)\rangle\right\} \nu(\mathrm{d}x),$$

**Derivatives** of the log partition function $\mathcal{A}(\eta)$ yield cumulants

$$\nabla \mathcal{A}(\eta) = \mathbb{E}\left[t(x)\right], \qquad \nabla^2 \mathcal{A}(\eta) = \mathbb{E}\left[t(x)t(x)^{\mathsf{T}}\right] - \mathbb{E}\left[t(x)\right]\mathbb{E}\left[t(x)\right]^{\mathsf{T}},$$

Compound models' statistics are **polynomials** in component statistics

$$\log p(z_1, z_2, \ldots, z_M, x) = \textstyle\sum_{\beta\in\mathcal{B}} \langle \eta_\beta(x),\, t_{z_1}(z_1)^{\beta_1} \otimes \cdots \otimes t_{z_M}(z_M)^{\beta_M}\rangle \quad (3)$$
$$\triangleq g(t_{z_1}(z_1), \ldots, t_{z_M}(z_M)),$$

## Too much math for a poster

When $g$ is multi-linear (has max-degree 1), then

**Claim 2.1.** *Given an exponential family with density of the form* (3), *we have*

$$p(z_m \,|\, z_{\neg m}) = \exp\left\{\langle \eta^*_{z_m},\, t_{z_m}(z_m)\rangle - \mathcal{A}_{z_m}(\eta^*_{z_m})\right\} \;\; \text{where} \;\; \eta^*_{z_m} \triangleq \nabla_{t_{z_m}} g(t_{z_1}(z_1), \ldots, t_{z_M}(z_M)).$$

Define a variational family using the same component statistics

$$q(z) = \prod_m q(z_m; \eta_{z_m}), \qquad q(z_m; \eta_{z_m}) = \exp\left\{\langle \eta_{z_m},\, t_{z_m}(z_m)\rangle - \mathcal{A}_{z_m}(\eta_{z_m})\right\}, \quad (4)$$

$$\log p(x) = \log \int p(z, x)\, \nu_z(\mathrm{d}z) = \log \mathbb{E}_{q(z)}\left[\frac{p(z,x)}{q(z)}\right] \geq \mathbb{E}_{q(z)}\left[\log \frac{p(z,x)}{q(z)}\right] \triangleq \mathcal{L}. \quad (5)$$

**Claim 2.2.** *Given a model with density of the form* (3) *and variational problem* (4)-(5), *we have*

$$\arg\max_{\eta_{z_m}} \mathcal{L}(\eta_{z_1}, \ldots \eta_{z_M}) = \nabla_{\mu_{z_m}} g(\mu_{z_1}, \ldots, \mu_{z_M}) \;\; \text{where} \;\; \mu_{z_{m'}} \triangleq \nabla \mathcal{A}_{z_{m'}}(\eta_{z_{m'}}), \;\; m' = 1, \ldots, M.$$

## A general view on conjugacy: punchlines

- When energy is a **multi-linear polynomial** in **tractable statistic functions**…
  - Generic **Gibbs** via autodiff and a sampler for each statistic
  - Generic **structured mean field** and **SVI** via autodiff and a log normalizer for each statistic
  - Generic **marginalization** via autodiff and a log normalizer for each statistic
- Can write **generic implementations** of **structure-exploiting algorithms**…
  - but only once we're given the **polynomial representation**
  - … and those are hard to write directly!
- Find polynomial representations automatically?

## Term rewriting problem statement

Given a Python function denoting $f : \mathbb{R}^n \mapsto \mathbb{R}$ that has a representation

$$f = g \circ h \qquad \text{for a multi-lin. polynomial} \qquad g : \mathbb{R}^{n_1} \times \cdots \times \mathbb{R}^{n_M} \to \mathbb{R},$$

where the coordinate functions $h = (h_1, \ldots, h_M)$ come from a known set,

1. identify each $h_m$, and
2. produce a Python function to evaluate $g$.

## Domain-specific term graph rewriting implementation

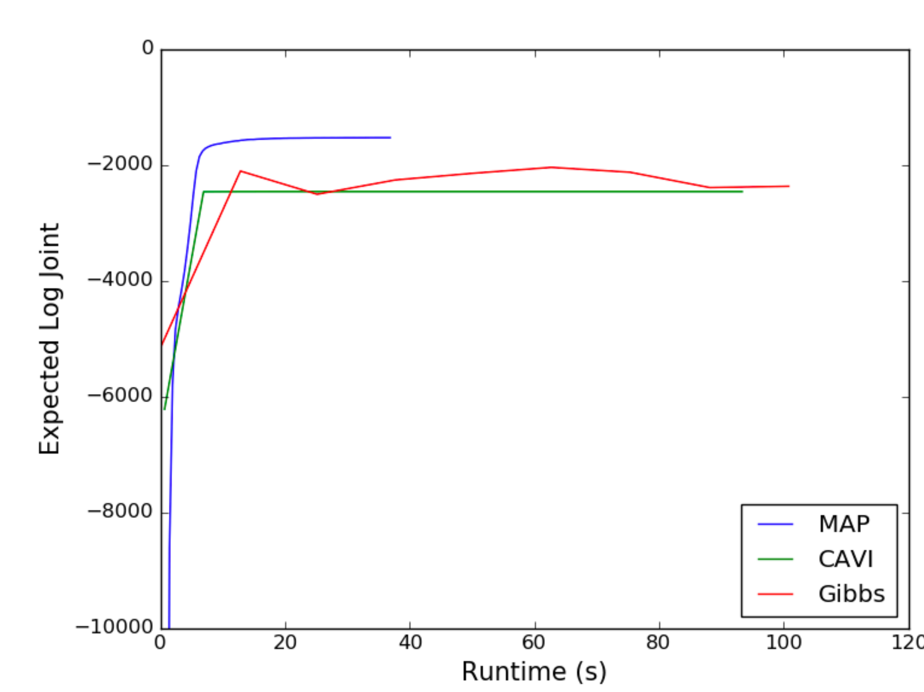
- **Tracer** using Autograd's API to map Python to term graphs
- **Pattern matcher** to do pattern-directed invocation
  - Python-embedded pattern language
  - Compiled into **continuation-passing matcher combinators** (~300 loc)

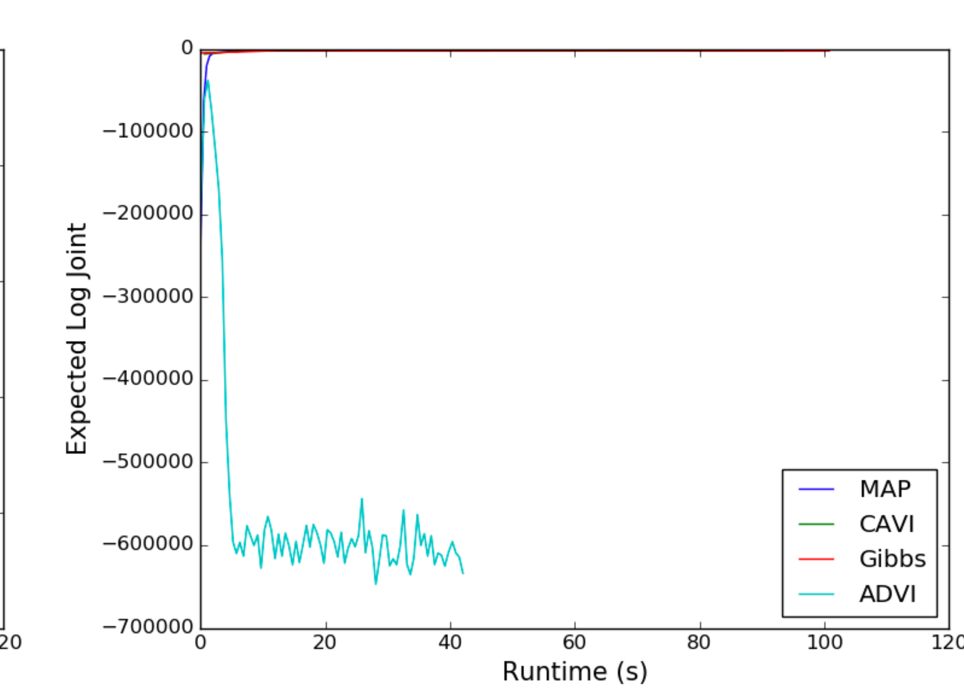
```
pat = (Einsum, Str('formula'), Segment('args1'),
       (Choice(Subtract('op'), Add('op')), Val('x'), Val('y')), Segment('args2'))
```

- **Rewriters** are syntactic graph macros using tracing to get **quasi-quasiquotes**

```
def rewriter(formula, op, x, y, args1, args2):
  return op(np.einsum(formula, *(args1 + (x,) + args2)),
            np.einsum(formula, *(args1 + (y,) + args2)))

distribute_einsum = Rule(pat, rewriter) # Rule is a namedtuple
```

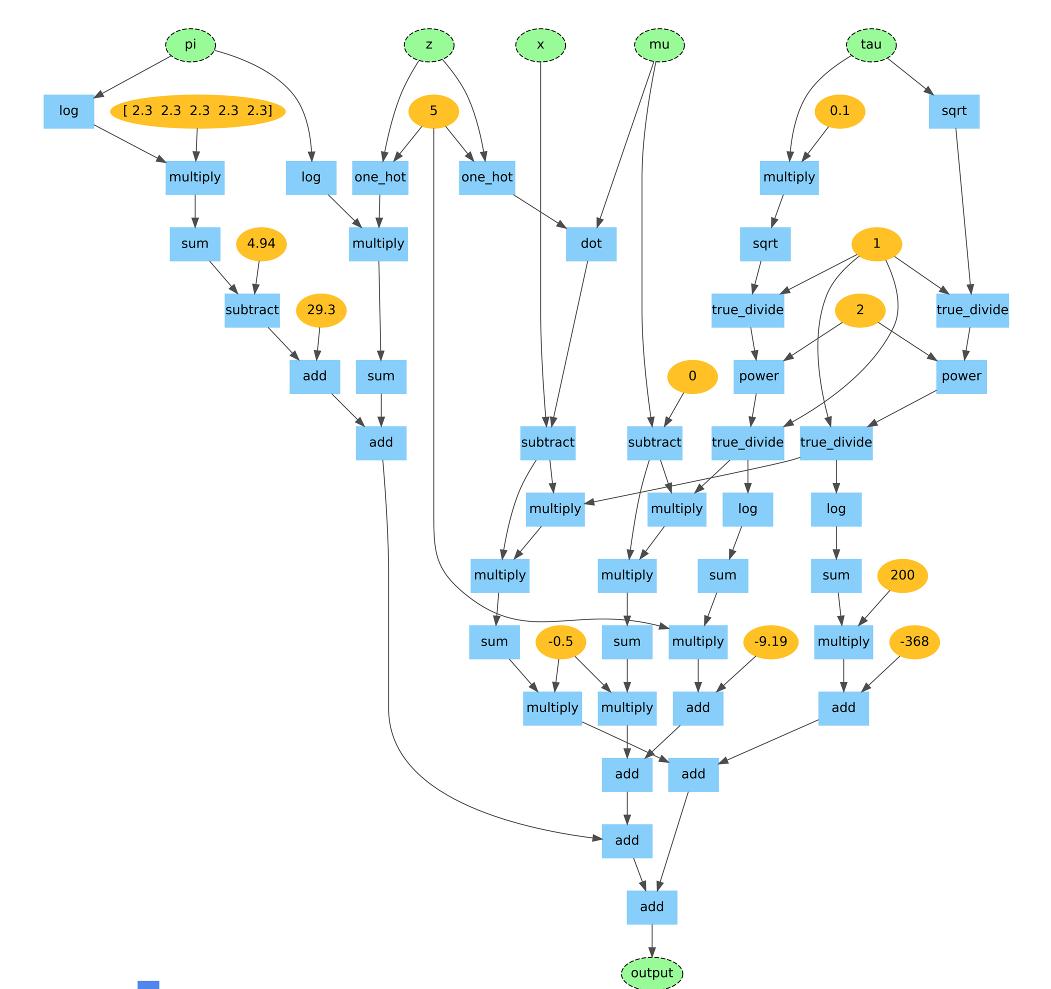
```
supports = (SIMPLEX, INTEGER, REAL, NONNEGATIVE)
g, As = multilin_repr(log_joint, example_vals, supports)
```



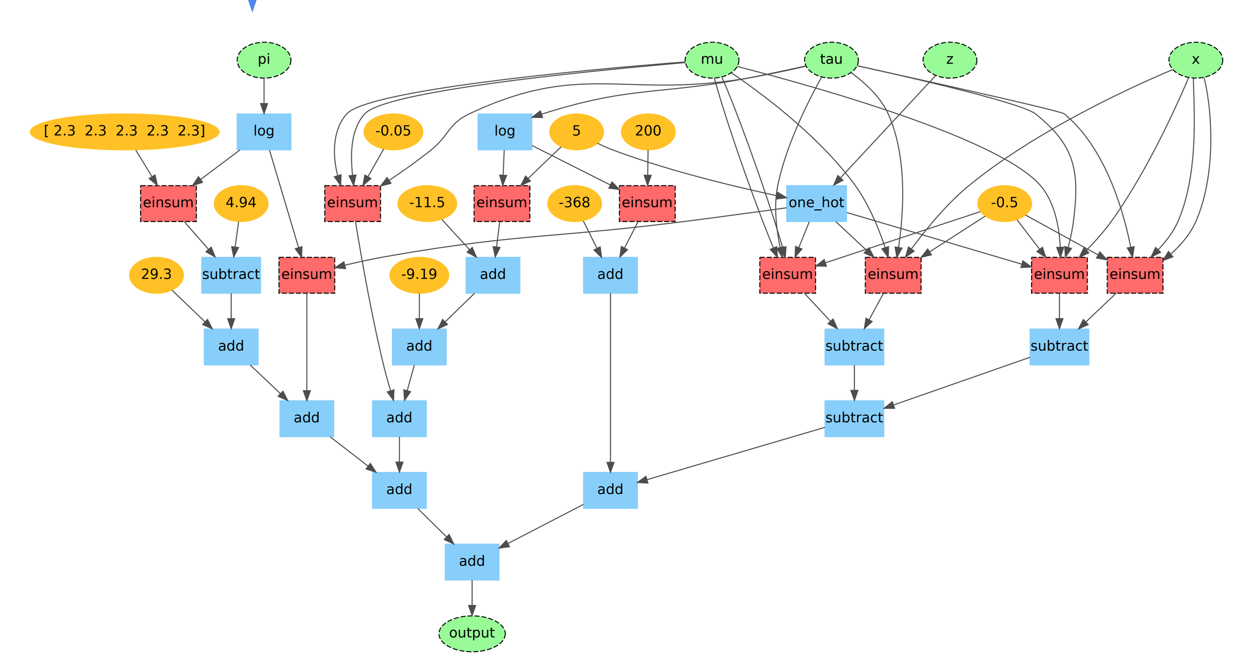




```
def normal_logpdf(x, loc, scale):
  prec = 1. / scale**2
  return -(np.sum(prec * mu**2) - np.sum(np.log(prec))
           + np.log(2. * np.pi)) * N / 2.

def log_joint(pi, z, mu, tau, x):
  logp = (np.sum((alpha-1)*np.log(x))
          - np.sum(gammaln(alpha))
          + np.sum(gammaln(np.sum(alpha, -1))))
  logp += normal_logpdf(mu, 0., 1./np.sqrt(kappa * tau))
  logp += np.sum(one_hot(z, K) * np.log(pi))
  logp += ((a-1)*np.log(tau) - b*tau + a*np.log(b)
           - gammaln(a))

  mu_z = np.dot(one_hot(z, K), mu)
  loglike = normal_logpdf(x, mu_z, 1./np.sqrt(tau))

  return logp + loglike
```

1 **Trace** log joint density given example values and supports



2 **Rewrite** term graph to expose exponential family structure



3 Generic implementations of mean field, marginalization, Gibbs, etc. (in plain Python!)