AutoPPL

James Yang, Jacob Austin, Jenny Chen, Lucie le Blanc

What Is AutoPPL?

Probabilistic programming language

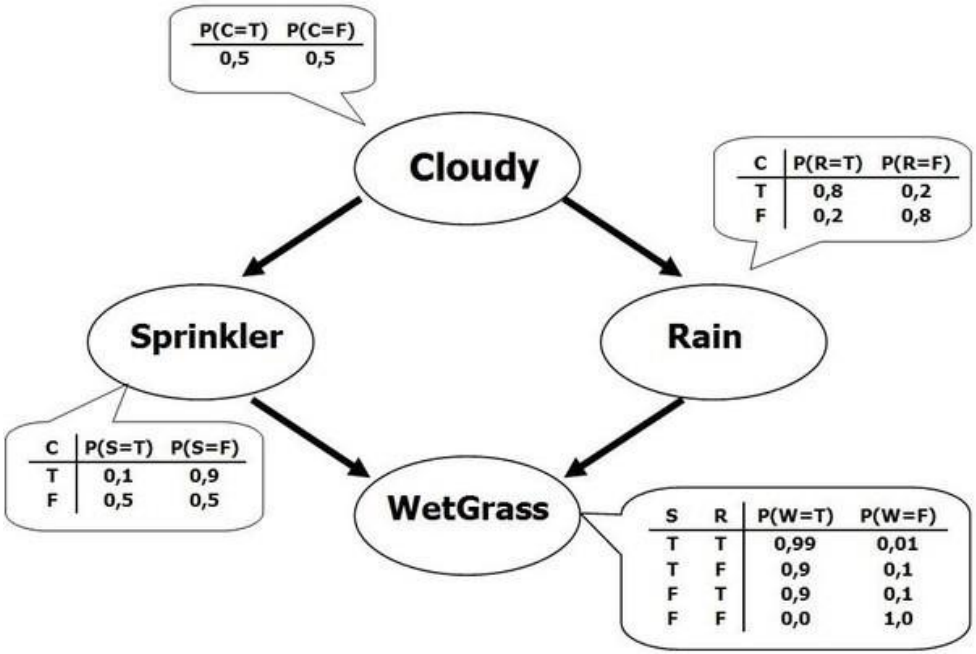
A way to view the world through Bayesian models

Core statistics library

High-performance header-only library using C++ concepts

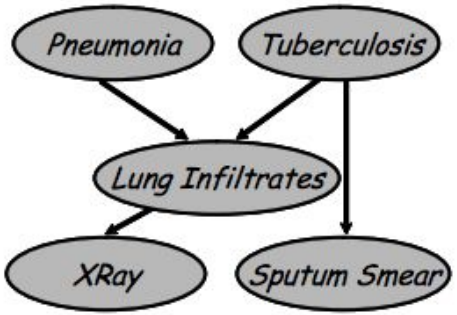
The Bayesian worldview

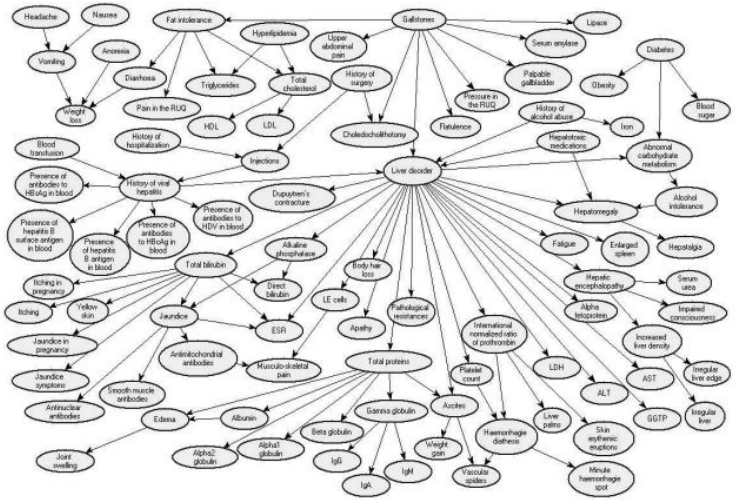
- The world is **probabilistic** and has **causal relationships.**
- Can reason about causes and hidden relationships using **generative models**!
- Given a model of the world and some observations, can we reason about things we don't see? Can we automate this?



Applications

- Since 1900s, Bayesian statistics and graphical models have been used in physics, probability theory, and genetics.
- Often used in medical diagnoses, speech recognition/NLP (as HMMs), causal inference, computer vision, modeling protein & genetic structures





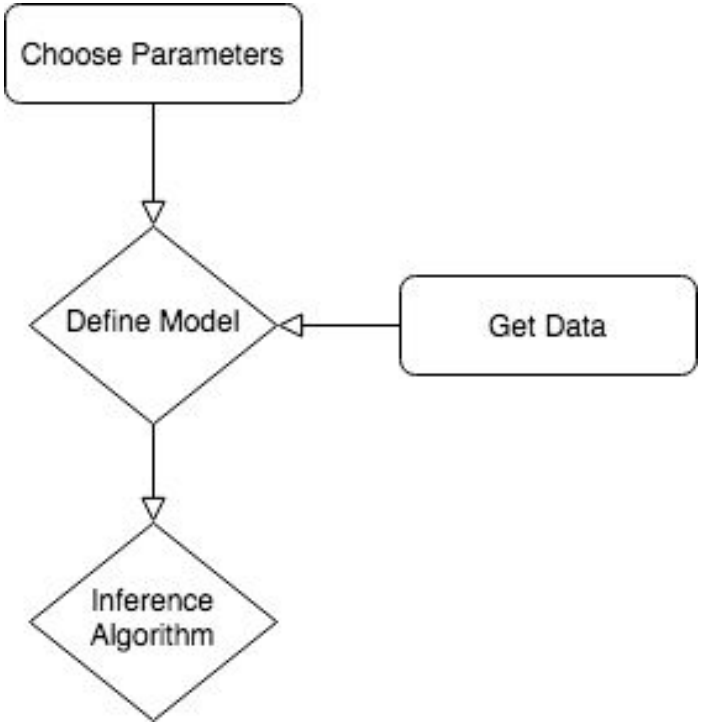
What does a probabilistic programming language do?

How do probabilistic programming languages come in?

- Easy language for describing generative models.
- **Fast, automated inference** on these models.

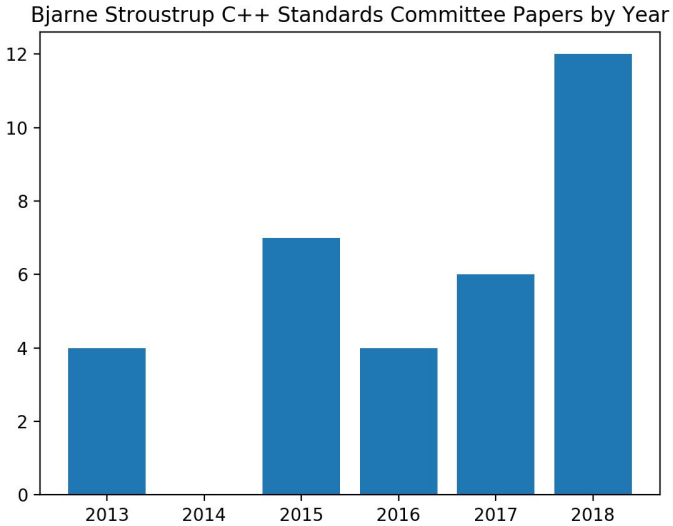
What kind of inferences do we want to make?

- Probability of an observed outcome under **many different models.**
- Most likely values for **hidden variables** under your model given observations.



Hello World (Bjarnian Regression)

Have you ever wondered how many C++ standards committee papers Prof. Stroustrup is going to publish next year? Let's try to model it.



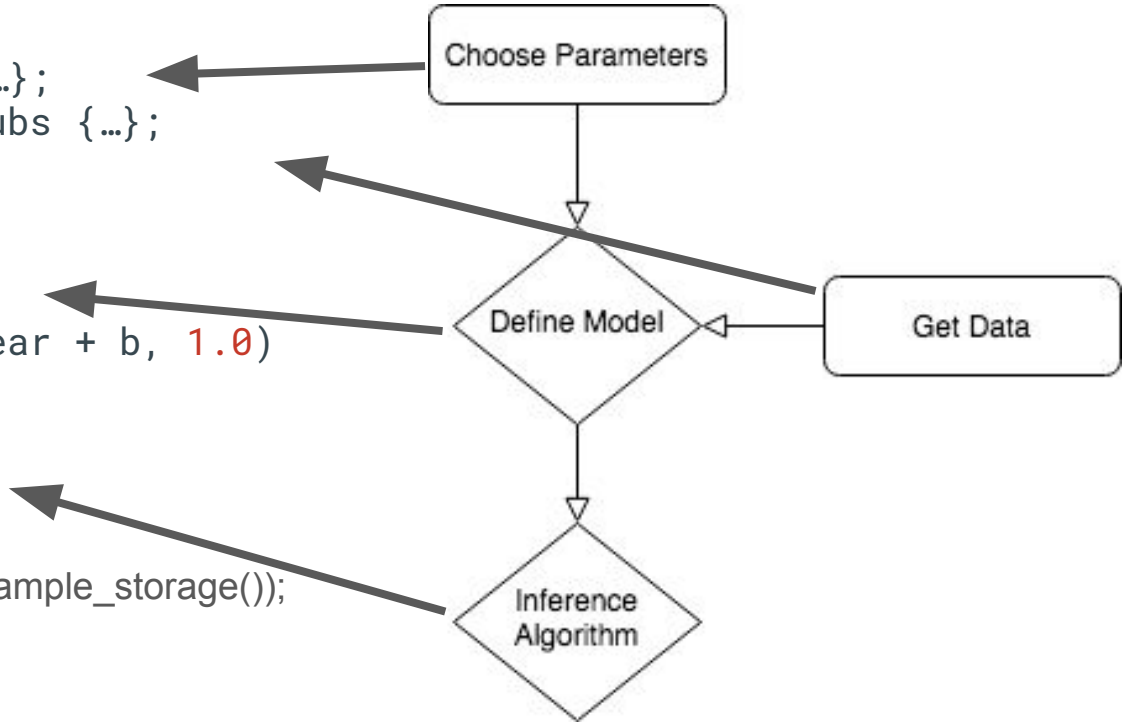
Wow he's pretty prolific! That looks pretty linear, doesn't it? But there's definitely some noise. What if we assume the number of publications is **approximately linear, with some Gaussian noise**, i.e.

publications ~ Normal(wx + b, 1)

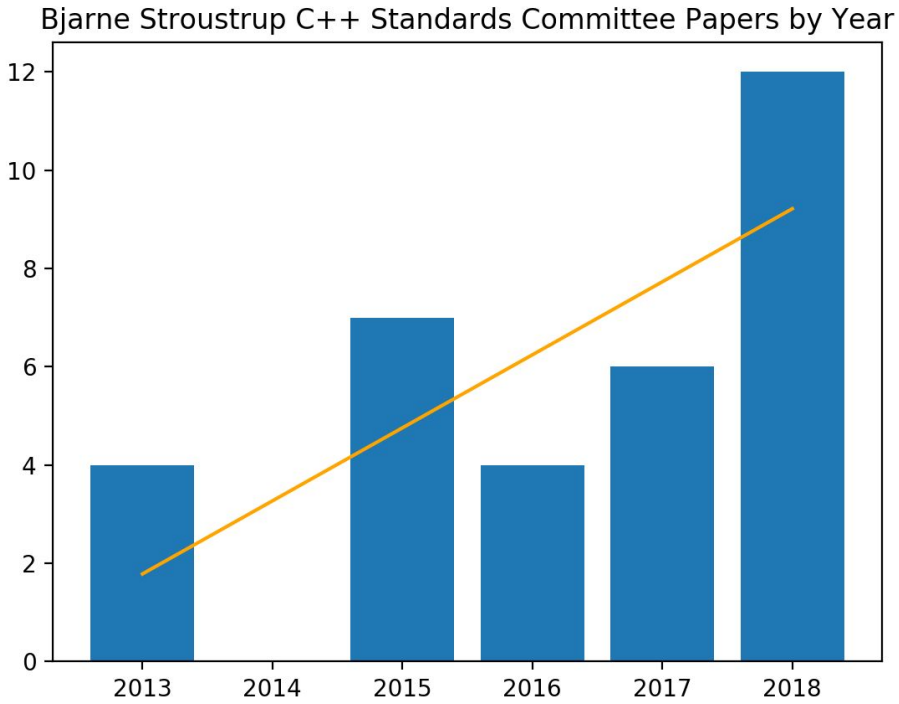
for some w and some b? Can we fit this?

How we do this in AutoPPL?

```
ppl::Param<double> w {…}, b {…};
ppl::Data<double> year {…}, pubs {…};

auto model = (
    w |= ppl::normal(0., 10.),
    b |= ppl::normal(0., 10.),
    pubs |= ppl::normal(w * year + b, 1.0)
);

ppl::mh(model, num_samples);

auto mean_w = sample_mean(w.get_sample_storage());
```



Bjarnian Regression Results



Predicted (mean) w: 1.19672131
Predicted (mean) b: -2406.307

So in 2021, Prof. Stroustrup will publish about 12 standards committee papers!

Big library constructs: Variables

ppl::Data<T> and ppl::Param<T>

- ppl::Data holds observed data, ppl::Param is a hidden variable to be estimated.
- ppl::Param is usually initialized with Param { T * storage_ptr } for sample storage (does not own its storage)
- Each satisfy difference concepts: “data concept” and “parameter concept”.

```
ppl::Data<int> x  {1, 2, 3};
x.observe(4); // adds a fourth observation
to x
x.get_value(3); // get the 4th value in x
x.size();  // return 4
x.clear(); // empty variable
```

```
std::array<double, 100> storage;
// binds to storage to store samples
ppl::Param<double> w { storage.data() };
w.get_value(); // get the current value;
w.size();  // return 1
w.observe(); // compile-time error
```

Big library constructs: Algorithms

ppl::mh and ppl::nuts

- Metropolis-Hastings (MH) and No-U-Turn Sampler (NUTS)
- High-performance MCMC sampling algorithms
- Sample from the posterior distribution of a model.

```
ppl::Param<double> w { storage_ptr };
… // define model here

ppl::nuts(model, burn_samples, num_samples, n_adapt); // runs MCMC algorithm
on model, stores sampled parameters in model storage pointer.

double sum = std::accumulate(storage_ptr, storage_ptr + num_samples, 0);
auto avg_w = sum / num_samples;  // can average samples from nuts algorithm.
```

Big library constructs: Distributions

ppl::normal, ppl::uniform, ppl::bernoulli

- Many standard probability distributions.
- Each supports dist.pdf(value_t), dist.sample(Generator && gen), and automatic differentiation functions that support the NUTS sampler.

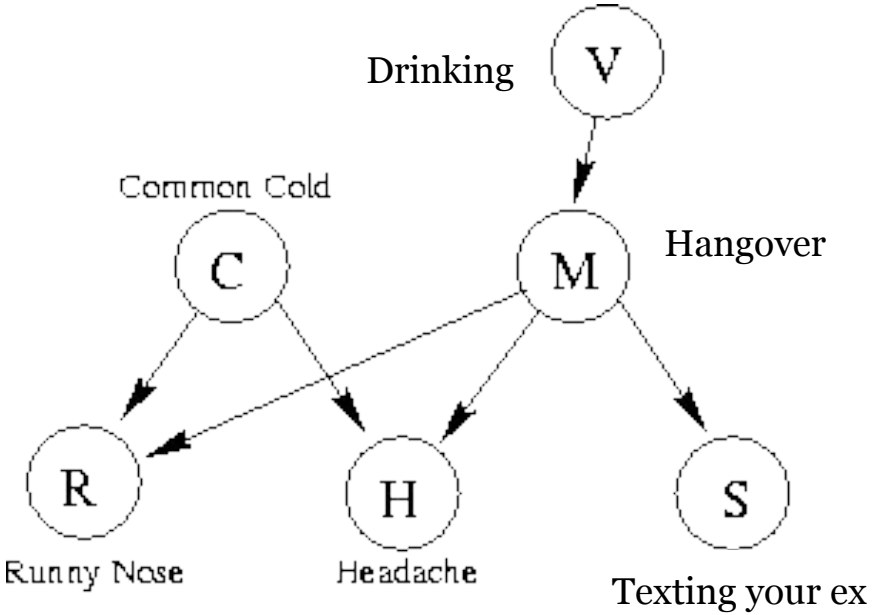
```
EXPECT_NEAR(norm.pdf(-10.231), 1.7267525955883482167 42E-15, tol);
EXPECT_DOUBLE_EQ(norm.log_pdf(-10.231),
                 std::log(1.7267525955883482167 42E-15));

std::random_device rd{};
std::mt19937 gen{rd()};

for (size_t i = 0; i < sample_size; i++) {
    sample[i] = norm.sample(gen);
}
```

Diagnosing Cold or Hangover!

Let’s diagnose a common problem and formalize this in AutoPPL!

Let’s say **we observe R, H, and S** (symptoms). We want to **predict the probability of C, M, and V** given these dependencies.

We’ll make up some numbers for the probabilities, e.g. P(hangover | drinking) is 0.8 if you were hungover, and 0.01 otherwise.



Our model:

```
ppl::Param<int> drinking, common_cold, hangover;
ppl::Data<int> runny_nose, headache, texting_ex;

auto model = (
    drinking |= ppl::bernoulli(0.1),
    common_cold |= ppl::bernoulli(0.3),
    hangover |= ppl::bernoulli(0.8 * drinking + 0.01 * (1 - drinking)),
    runny_nose |= ppl::bernoulli(0.3 * common_cold + 0.4 * hangover +
0.1),
    headache |= ppl::bernoulli(0.5 * common_cold + 0.4 * hangover + 0.1),
    texting_ex |= ppl::bernoulli(0.9 * hangover)
);
```

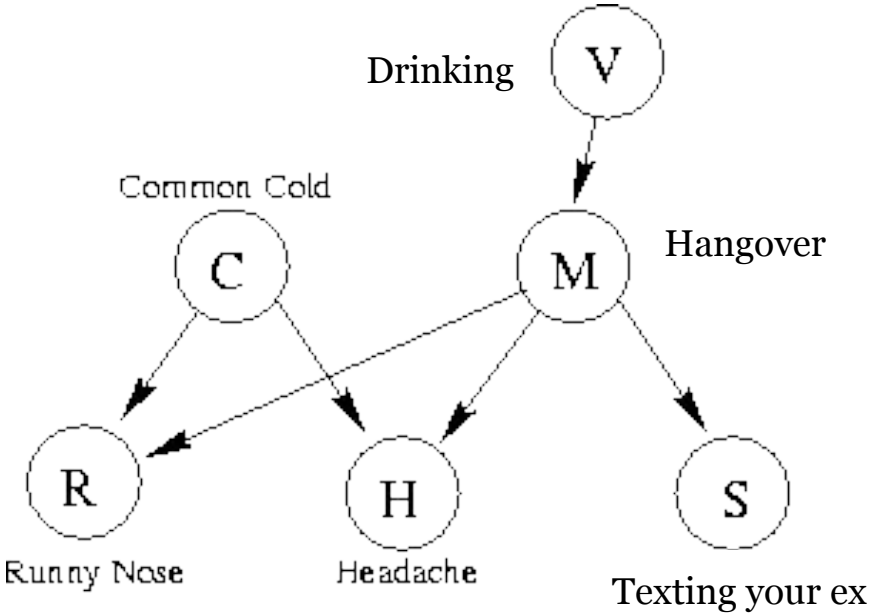
Inference:

```
runny_nose.observe(0.0);  // what we observe
headache.observe(0.0);
texting_ex.observe(1.0);

ppl::mh(model, samples);  // sample from the distribution

std::cout << "drinking: " << average(drinking_samples) << std::endl;
std::cout << "common cold: " << average(cold_samples) << std::endl;
std::cout << "hangover: " << average(hangover_samples) << std::endl;
```

Results:

P(hangover) = 1
P(drinking) = 0.89
P(cold) = 0

which makes sense, since we texted our ex, but we didn't have a runny nose or a headache. This agrees with the analytic results (the true analytic results are 1, 0.898876404, 0)



Benchmarks & Performance

Data: draw 100 samples of Y from N(0, 1)

Model:

```
Y ~ N(l1 + l2, sigma)
l1 ~ N(0, 10)
l2 ~ N(0, 10)
sigma ~ Uniform(0, 20)
```

AutoPPL

pure C++ library

STAN

Python bindings for underlying C++ library

PyMC3

pure Python library

PyMC3

```
y_data = np.random.normal(size = 100)

basic_model = pm.Model()
with basic_model:
    sigma = pm.Uniform('sigma', 0, 20)
    l1 = pm.Normal('l1', 0, 10)
    l2 = pm.Normal('l2', 0, 10)
    y = pm.Normal('y', mu=(l1 + l2),
            sigma=sigma, observed=y_data)

with basic_model:
    data = pm.sample(draws=1000, n_init=1000,
                     chains=1, cores=1, tune=1000)
```

STAN

```
my_model_code = """
data {
 int N;
 real y[N];
}
parameters {
 real lambda1;
 real lambda2;
 real<lower=0>
sigma;
}
transformed parameters {
 real mu;
 mu = lambda1 + lambda2;
}
model {
 sigma ~ uniform(0, 20);
 lambda1 ~ normal(0, 10);
 lambda2 ~ normal(0, 10);
 y ~ normal(mu, sigma);
}
"""
```

STAN

```
N = 100
y = np.random.normal(size = N)
my_model_dat = { 'N' : N, 'y' : y }

sm = pystan.StanModel(model_code=my_model_code)
fit = sm.sampling(data=my_model_dat,
                  chains=1, n_jobs=1)
```

AutoPPL

```
std::normal_distribution n(0.0, 1.0);
std::mt19937 gen(0);
ppl::Data<double> y;

ppl::Param<double> lambda1, lambda2, sigma;
auto model = (
    sigma |= ppl::uniform(0.0, 20.0),
    lambda1 |= ppl::normal(0.0, 10.0),
    lambda2 |= ppl::normal(0.0, 10.0),
    y |= ppl::normal(lambda1 + lambda2, sigma)
);
```

AutoPPL

```
for (int i = 0; i < 100; i++) {
    y.observe(n(gen));
}

std::array<double, 1000> l1_storage,
                         l2_storage, s_storage;
lambda1.set_storage(l1_storage.data());
lambda2.set_storage(l2_storage.data());
sigma.set_storage(s_storage.data());

ppl::nuts(model, 1000, 1000, 1000);
```

Benchmarks & Performance

Model: (100 samples)

```
Y ~ N(l1 + l2, sigma)
l1 ~ N(0, 10)
l2 ~ N(0, 10)
sigma ~ Uniform(0, 20)
```

AutoPPL

```
Compilation time: 3.39s
Run time: 0.987008s
```

STAN

```
Compilation time: 40.48s
Run time: 0.613672s (Warm-up)
          0.704271s (Sampling)
          1.31794s (Total)
```

PyMC3

```
Run time: 38s
```

Sampling Results

STAN

	mean	sd
lambda1	0.3	7.2
lambda2	0.05	7.2
sigma	1.03	0.07

	mean	sd
lambda1	0.42	6.89
lambda2	-0.47	6.89
sigma	0.98	0.07

AutoPPL

	mean	sd
lambda1	-1.18	6.78
lambda2	1.28	6.75
sigma	0.95	0.07

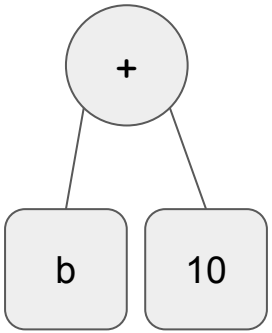
	mean	sd
lambda1	-1.35	6.98
lambda2	1.45	6.99
sigma	0.95	0.07

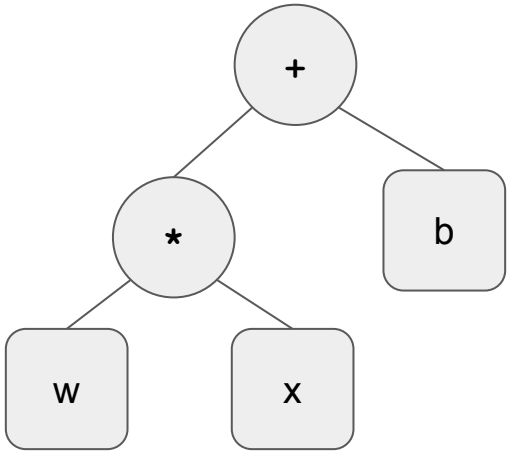
Technical/implementation details:

- Variable Expressions
- Distribution Expressions
- Model Expressions
- Model memory layout
- Concepts
- Supported Features

Variable Expressions

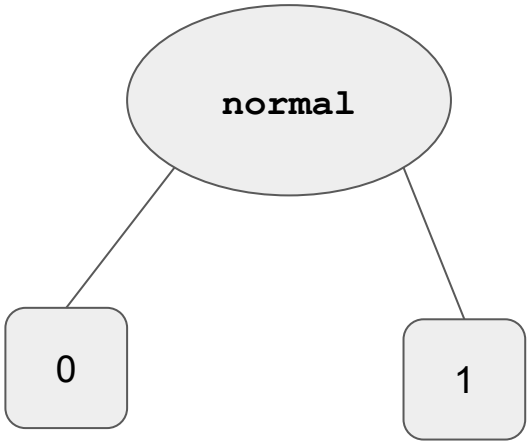
- Arithmetic expression over data or parameters
- E.g. w * x + b, where w,b are ppl::Param<T> and x is ppl::Data<T>
- Constructs a tree at compile-time to lazily evaluate the expression
- Currently supporting binary operators (+,-,*,/)

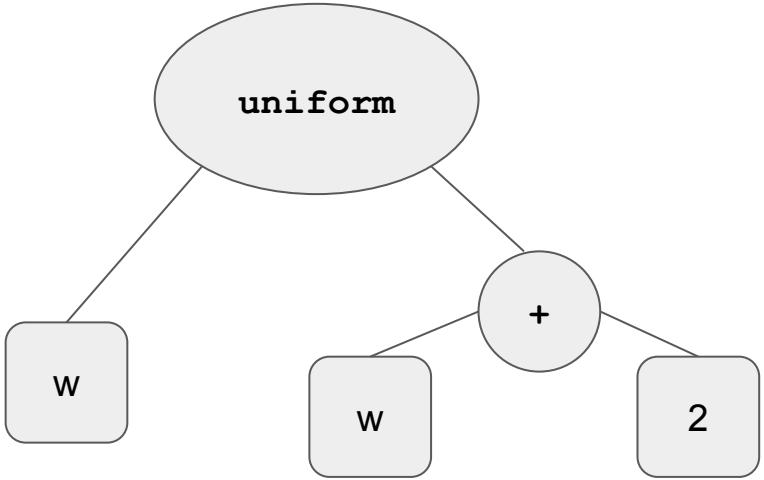




Distribution Expressions

- Examples: ppl::normal(0., 1.), ppl::uniform(w, w + 2)
- Distribution expression can store any expressions for its parameters





Model Expressions

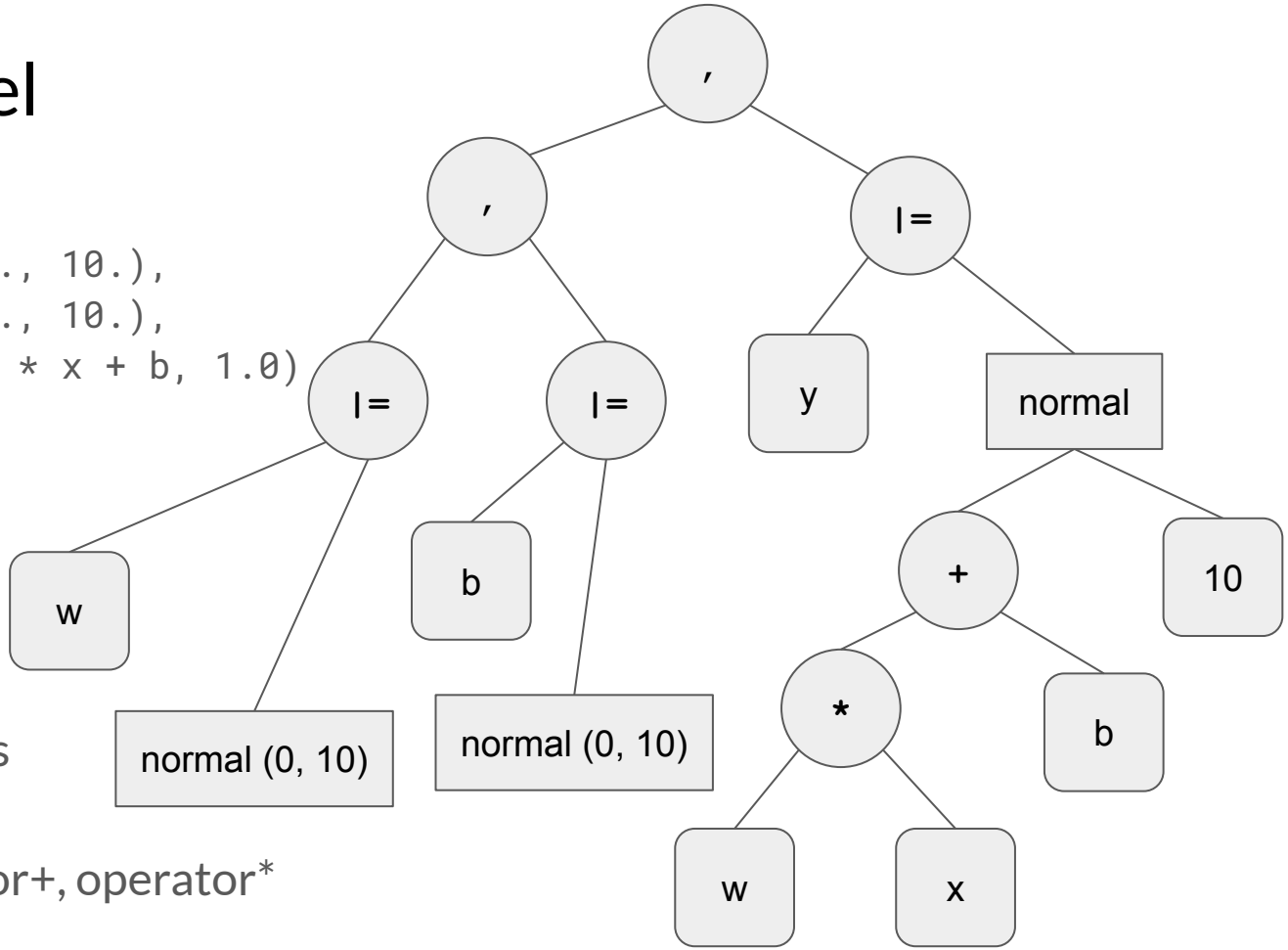
- Relate variables with distribution expressions.
- Complicated type because of expression templates, but not needed by users
- Syntax: var |= distribution(params);
- Very small
  - sizeof models with about 10 layers and non-trivial dependence is typically less than 500 bytes
  - acts as a viewer over existing data
  - advantage: can have many models over the same data

```
auto model1 = (
        w |= ppl::normal(0., 10.),
        b |= ppl::normal(0., 10.),
        y |= ppl::normal(w * x + b, 1.0)
    );
// one model for w,b,x,y
```

```
auto model2 = (
        w |= ppl::uniform(-10., 10.),
        b |= ppl::uniform(-10., 10.),
        y |= ppl::normal(w * x + b, 1.0)
    );
// another model for the same w,b,x,y
```

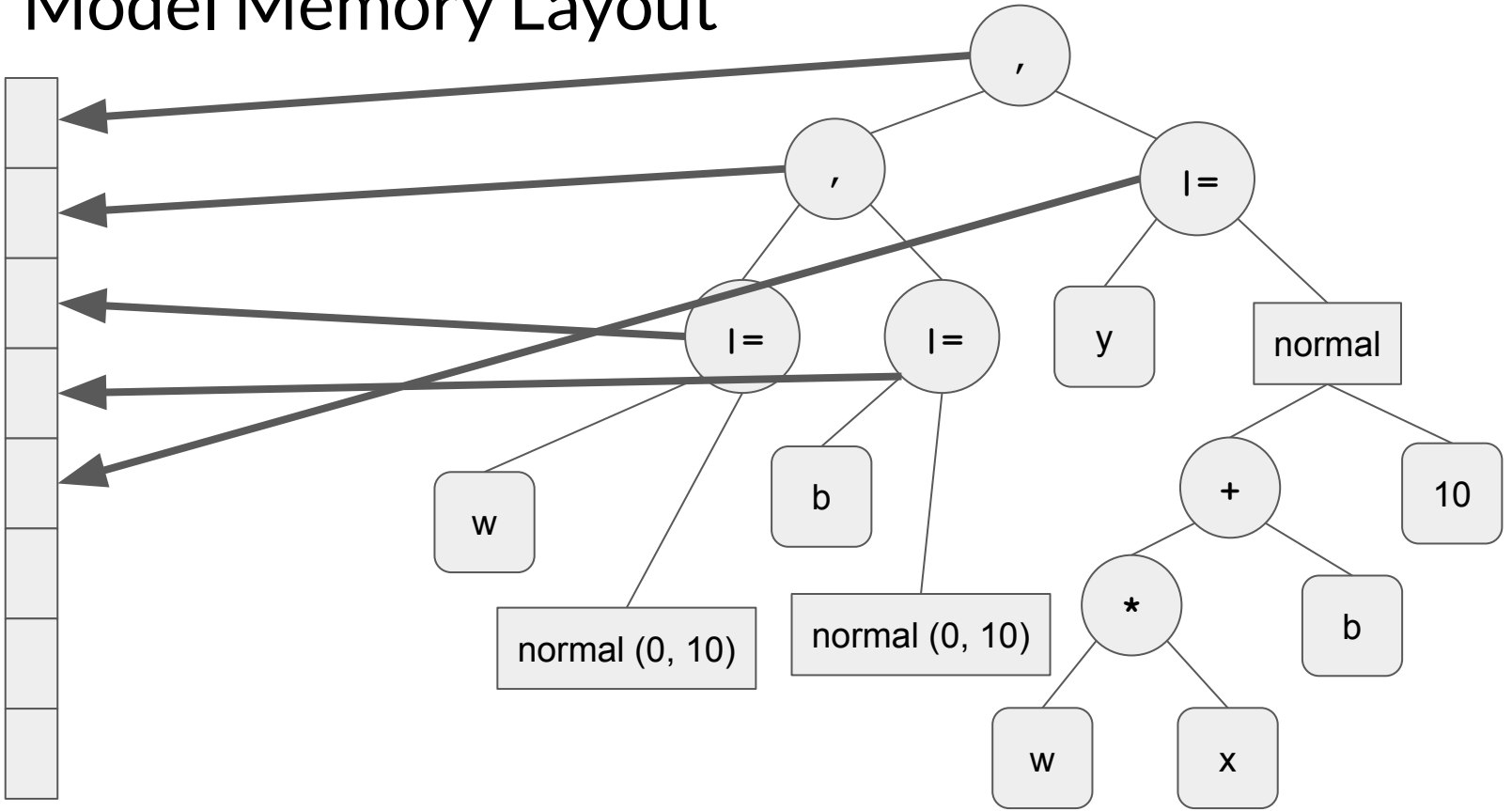
Building a model

```
auto model = (
    w |= ppl::normal(0., 10.),
    b |= ppl::normal(0., 10.),
    y |= ppl::normal(w * x + b, 1.0)
);
```



- Built on the stack at compile time
- Overload operators such as operator, operator|=, operator+, operator*

Model Memory Layout



Sampling Algorithms

- Model expressions can be used to compute joint PDF
- In fact, joint PDF is all we need to sample from posterior distribution

- Metropolis-Hastings
- No-U-Turn Sampler (NUTS)
  - Requires automatic differentiation (FastAD) to use gradient information
  - Takes model expression and creates an AD expression at compile-time
  - AD expression is also a contiguous chunk of memory (stack-allocated)

Concepts in C++17

- For portability and abiding the current standard, we made a homegrown concepts feature using template metaprogramming.
- Neat feature: compiler-error explicitly says which member alias or member function is missing

```
template <class T>
 inline constexpr bool
assert_is_var_expr_v =
     assert_var_expr_is_base_of_v<T> &&
     details::assert_is_not_var_v<T> &&
     assert_has_type_value_t_v<T> &&
     assert_has_func_get_value_v<const T>
     ;
```

```
In file included from
/Users/jhyang/sandbox/autoppl/test/util/var_expr_tr
aits_unittest.cpp:2:
In file included from
/Users/jhyang/sandbox/autoppl/include/autoppl/util
/var_expr_traits.hpp:3:
/Users/jhyang/sandbox/autoppl/include/autoppl/util
/concept.hpp:215:1: error: static_assert failed
      "Type does not have public,
non-overloaded member function get_value"
DEFINE_HAS_FUNC(get_value);
^~~~~~~~~~~~~~~~~~~~~~~~~~
```

Why go through such trouble for C++17 Concepts?

- Ex. defined operator overloads such as `operator-` with **very rudimentary** constraints
- Other libraries also overload such operators and use similar type aliases
- Weird bug:
  - Used `std::chrono::duration` objects to compute current time since epoch:
    - `std::chrono::duration_cast<std::chrono::millisecond>(start - end)`
  - Operator- overloaded in chrono library!
  - Compiler kept choosing **our** overload!
    - Duration objects fit our rudimentary constraints
- Need a more rigorous way to constrain types

Supported Features

- Data, Params
- Construct model expressions
- Distributions
  - Uniform
  - Gaussian
  - Bernoulli
  - Discrete
- Sampling Algorithms
  - MH (Metropolis-Hastings)
  - NUTS (No-U-Turn Sampler)

Testing, Benchmark, CI

- GoogleTest
  - Currently, about 110 unit tests
- Google Benchmark
  - Benchmarking against most commonly used libraries (STAN, PyMC3)
- Continuous Integration (CI)
  - Travis CI
  - Currently supporting machine: Ubuntu 14.04
  - Currently supporting compilers: g++-7, g++-8, g++-9
  - Check test coverage using Coveralls

Third Party Tools

- Armadillo: matrix library used for inference algorithms.
- Clang: one of the main compilers used.
- CMake: build system.
- Coveralls: check test coverage.
- Cpp Coveralls: check test coverage specifically for C++ code.
- FastAD: automatic differentiation library.
- GCC: one of the main compilers used.
- Google Benchmark: benchmark library algorithms.
- GoogleTest: unit/integration-tests.
- Travis CI: continuous integration for Linux using GCC.
- Valgrind: check memory leak and errors.

Plans for the Future

- Supporting more distributions
- Integrating with auto-differentiation libraries (or improving FastAD) to implement optimization based algorithms (e.g. MAP estimation, other HMC algorithms like NUTS)

Live Demo

Thank You!

Thank you to Prof. Stroustrup and the TAs for a wonderful semester!