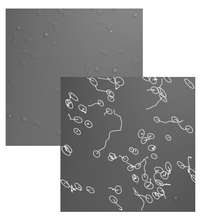
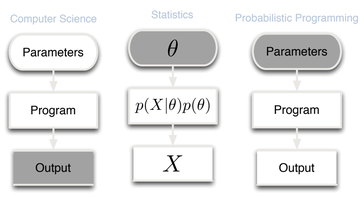
Unique Qualifications And Capabilities

- Know-How
  - Neuroscience
    - Black, Wood, Wu, Method and System for automatic decoding of motor cortical activity, US Patent, 2005.
    - Gasthaus, Wood, Görür, Teh, Dependent Dirichlet process spike sorting, 2009
  - Machine Learning
    - Neiswanger, Wood, Xing, The Dependent Dirichlet Process Mixture of Objects for Detection-free Tracking and Object Modeling, 2014
    - Caron, Neiswanger, Wood, Doucet, Davy, Generalized Pólya Urn for Time-Varying Pitman-Yor Processes, In Submission 2014
  - Probabilistic Programming
    - Wood, van de Meent, Mangsinghka, A New Approach to Probabilistic Programming Inference, AISTATS 2014
    - Paige, Wood, A Compilation Target for Probabilistic Programming Languages, ICML 2014
- People
  - Lab
  - 3YP student stream

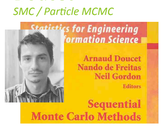




Teh
Bayesian nonparametrics

Zisserman
Computer vision

Doucet
SMC / Particle MCMC



Caron, Roberts, Vidaldi, Osborne, Newman, Posner, de Freitas, Blunsom, …

Research areas of interest

Anatomy
- Top-down biologically-parameterized, model-based regularization for tracking and structure discovery
- Efficient, massively parallel automatic inference

Function
- Automatic stochastic inversion of function simulators

Artificial Intelligence
- Expressive models and automatic inference via probabilistic programming

Probabilistic-C, a Probabilistic Programming Language @ICML 2014

Probabilistic C

Forward inference techniques such as sequential Monte Carlo [1] and particle Markov chain Monte Carlo [2] for probabilistic programming [3] can be implemented in any programming language by creative use of standardized operating system functionality including processes, forking, mutexes, and shared memory.

- Probabilistic C: a probabilistic programming intermediate representation language, which itself can be compiled to parallel machine code by standard compilers
- Standard C with two new directives: `observe` and `predict`
- Compiled programs automatically emit posterior samples of predicted quantities, conditioned on the observed data
- Freely incorporate existing libraries and C code, including black box simulators

Implementation

- The POSIX `fork` function causes a process to clone itself, creating a new process with an identical copy of the program execution state
  - Lazy copy-on-write procedure for duplicating memory contents: efficient
- Using `fork` we can branch and explore many downstream program execution paths
- Massively parallel: each downstream path is explored by an independent OS process
- Global synchronization is handled via shared memory segments

Usage

Simple example: iid draws from a Gaussian distribution with unknown mean.

$$\mu \sim \mathrm{Normal}(1,5)$$
$$y_1, y_2 \overset{iid}{\sim} \mathrm{Normal}(\mu, 2)$$

Sample from prior:

```
#include "probabilistic.h"

int main(int argc, char **argv) {

    double var = 2;
    double mu = normal_rng(1, 5);

    printf("%f\n", normal_rng(mu, var));
    printf("%f\n", normal_rng(mu, var));

    printf("mu,%f\n", mu);

    return 0;
}
```

Sample from posterior:

```
#include "probabilistic.h"

int main(int argc, char **argv) {

    double var = 2;
    double mu = normal_rng(1, 5);

    observe(normal_lnp(9, mu, var));
    observe(normal_lnp(8, mu, var));

    predict("mu,%f\n", mu);

    return 0;
}
```

Another example: a hidden Markov model, with Gaussian emission distributions.

$$z_0 \sim \mathrm{Discrete}([1/K,\ldots,1/K]) \quad z_n|z_{n-1} \sim \mathrm{Discrete}(T_{z_{n-1}}) \quad y_n|z_n \sim \mathrm{Normal}(\mu_{z_n}, \sigma^2)$$

Sample from prior:

```
#include "probabilistic.h"
#define K 3
#define N 11

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };



/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 };

/* Generative program for a HMM */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);

        if (n > 0) {
            printf("data[%d],%f\n", n,
                   normal_rng(state_mean[states[n]], 1));
        }
        printf("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```

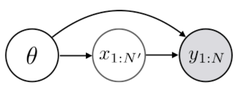
Sample from posterior:

```
#include "probabilistic.h"
#define K 3
#define N 11

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };

/* Observed data */
static double data[N] = { NAN, .9, .8,  .7,   0, -.025,
                         -5, -2, -.1,   0,  0.13 };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 };

/* Generative program for a HMM */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);

        if (n > 0) {
            observe(normal_lnp(data[n],
                               state_mean[states[n]], 1));
        }
        predict("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```

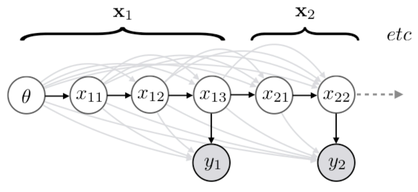
Inference

Define a probability distribution over program execution traces

- Enumerate all N observe statements, and the associated data points $y_1, \ldots, y_N$
- Enumerate all N' random choices made during the execution of the program $x_1, \ldots, x_{N'}$

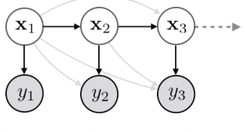


- Define a partition of the random choices such that each $\mathbf{x}_n$ contains all random choices made between observations y_{n-1} and y_n



- Run sequential Monte Carlo or particle MCMC algorithms over the resulting sequence of program execution states

$$p(y_{1:N}, \mathbf{x}_{1:N}) = \prod_{n=1}^{N} g(y_n|\mathbf{x}_{1:n}) f(\mathbf{x}_n|\mathbf{x}_{1:n-1})$$



- Note we only need to be able to simulate from $f(\mathbf{x}_n|\mathbf{x}_{1:n-1})$ and evaluate $g(y_n|\mathbf{x}_{1:n})$

Algorithm Details

Algorithm 1 Parallel SMC program execution

Assume: N observations, L particles
- launch L copies of the program (parallel)
- **for** $n = 1 \ldots N$ **do**
  - wait until all L reach `observe` y_n (barrier)
  - update unnormalized weights $\tilde{w}_n^{1:L}$ (serial)
  - **if** $ESS < \tau$ **then**
    - sample number of offspring $O_n^{1:L}$ (serial)
    - set weight $\tilde{w}_n^{1:L} = 1$ (serial)
    - **for** $\ell = 1 \ldots L$ **do**
      - fork or exit (parallel)
    - **end for**
  - **else**
    - set all number of offspring $O_n^{\ell} = 1$ (serial)
  - **end if**
  - continue program execution (parallel)
- **end for**
- wait until L program traces terminate (barrier)
- `predict` from L samples from $\hat{p}(\mathbf{x}_{1:N}^{1:L}|y_{1:N})$ (serial)

Algorithm 2 Parallel Particle Gibbs program execution

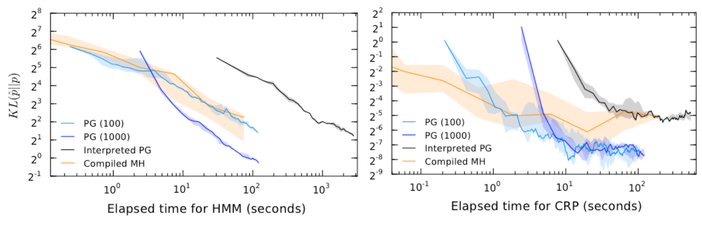
Assume: M iterations, N observations, L particles
- **for** $m = 1 \ldots M$ **do**
  - $L' \leftarrow L$ if $m = 1$, otherwise $L - 1$
  - launch S' copies of the program (parallel)
  - **for** $n = 1 \ldots N$ **do**
    - wait until all L' reach an `observe` (barrier)
    - compute weights for all particles (serial)
    - **if** $m > 1$ **then**
      - signal num offspring to retained trace (serial)
    - **end if**
    - **for** $\ell = 1 \ldots L'$ **do**
      - spawn retain / branch process [Algo. 3] (parallel)
    - **end for**
    - wait until L particles finish branching (barrier)
    - continue program execution (parallel)
  - **end for**
  - wait until L program traces terminate (barrier)
  - `predict` from L samples from $\hat{p}(\mathbf{x}_{1:N}^{1:L}|y_{1:N})$ (serial)
  - select and signal particle to retain (serial)
  - wait until N processes are ready to branch (barrier)
  - continue to next iteration (parallel)
- **end for**

Algorithm 3 Retain and Branch inner loop

Assume: input initial $C > 0$ children to spawn
- `is_retained` ← `false`
- **while** `true` **do**
  - **if** $C = 0$ and not `is_retained` **then**
    - discard this execution trace, exit
  - **else** $\{C \geq 0\}$
    - spawn C new children
  - **end if**
  - wait for signal which resets `is_retained`
  - **if** `is_retained` **then**
    - wait for signal which resets C
  - **else**
    - discard this execution trace, exit
  - **end if**
- **end while**

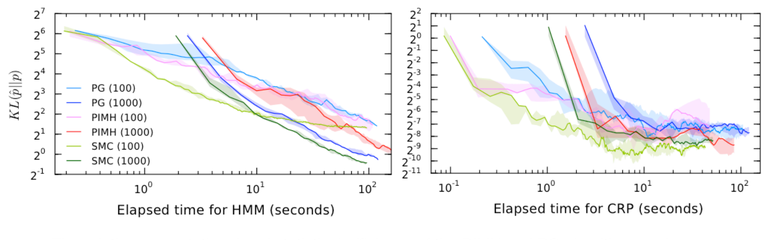
Experimental results

Performance compared to other probabilistic programming engines



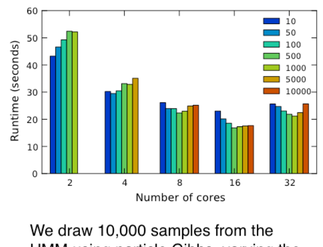
Convergence rates of probabilistic C, as compared to compiled Metropolis-Hastings [4] and interpreted particle Gibbs [5] implementations; compilation yields order-of-magnitude performance gains. Results shown for a hidden Markov model and a CRP mixture of Gaussians.

Performance across inference algorithms



The relative computational and statistical efficiency of the PIMH and PG algorithms varies across models and number of particles. Increasing the number of particles per iteration in the PMCMC algorithms involves a tradeoff between statistical efficiency and computational time per sweep.

Scalability across hardware



We draw 10,000 samples from the HMM using particle Gibbs, varying the number of particles each iteration, across five hardware configurations.

References

[1] Doucet, Arnaud, De Freitas, Nando, Gordon, Neil, et al. *Sequential Monte Carlo methods in practice.* Springer New York, 2001.

[2] Andrieu, Christophe, Doucet, Arnaud, and Holenstein, Roman. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[3] Goodman, Noah D., Mansinghka, Vikash K., Roy, Daniel M., Bonawitz, Keith, and Tenenbaum, Joshua B. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI2008)*, pp. 220–229, 2008.

[4] Wingate, David, Stuhlmueller, Andreas, and Goodman, Noah D. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, pp. 131, 2011.

[5] Wood, Frank, van de Meent, Jan Willem, and Mansinghka, Vikash. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, 2014.

Capabilities sought and types of research supported

Anatomy
- Sought : data and bottom-up feature extraction
- Support : top-down model-based regularization

Function
- Sought : stochastic circuit simulators
- Support : automatic inversion

Artificial Intelligence
- Sought : ?
- Support : programs that write programs, etc.

Frank Wood
Associate Professor
University of Oxford
fwood@robots.ox.ac.uk
http://www.robots.ox.ac.uk/~fwood/