Statistical Computing in the Big Data Context
(and something about LibBi 2.x)

Lawrence Murray

University of Oxford

5 October 2015
Introduction

Software engineering

▷ Requirements
▷ Design
▷ Implementation
▷ Verification
▷ Maintenance
Introduction

Software engineering
- Requirements
- Design
- Implementation
- Verification
- Maintenance

Statistical engineering?
- Prior elicitation
- Experimental design
- Data acquisition
- Model specification
- Inference
- Model validation
- Model selection
- Decision making

Lawrence Murray
University of Oxford
Statistical Computing in the Big Data Context (and something about LibBi 2.x)
Big data

- **Complex computing**
  Bigger computer architectures are also harder to deploy to.

- **Complex data**
  Data gets bigger, but also more complex (e.g. from multiple modalities), which means...

- **Complex models**
  ...to bring those modalities together, and...

- **Complex methods**
  ...to fit them.
Complexity

Models:

- Assemble complex models from simpler models.
- Object-oriented programming provides an example (encapsulation, inheritance, polymorphism).

Methods:

- Assemble complex methods from simpler methods?
- Functional programming provides an example?

Minimise redundancy, minimise opportunity for error, minimise incremental workload.
Parallelism

Data parallelism
  ▶ GPU computing
  ▶ SIMD computing

Task parallelism
  ▶ Multithreaded
  ▶ Multiprocess

In statistical computing, task parallelism is often used merely to implement data parallelism (e.g. MapReduce, partitioning of data sets).
The Zen of Python

*Tim Peters*

Beautiful is better than ugly. // Explicit is better than implicit. // Simple is better than complex. // Complex is better than complicated. // Flat is better than nested. // Sparse is better than dense. // Readability counts. // Special cases aren’t special enough to break the rules. // Although practicality beats purity. // Errors should never pass silently. // Unless explicitly silenced. // In the face of ambiguity, refuse the temptation to guess. // There should be one—and preferably only one—obvious way to do it. // Although that way may not be obvious at first unless you’re Dutch. // Now is better than never. // Although never is often better than *right* now. // If the implementation is hard to explain, it’s a bad idea. // If the implementation is easy to explain, it may be a good idea. // Namespaces are one honking great idea – let’s do more of those!
Simulation as specification?

- *Plug-and-play* models [Bretó et al., 2009].
- *Black-box* models.
- Probabilistic programming languages.
- Approximate Bayesian Computation.
- Simulation in pedagogy [Tintle et al., 2015].
LibBi

- **LibBi 1.x** was designed for Bayesian inference with state-space models, using Sequential Monte Carlo (SMC) and Particle Markov Chain Monte Carlo (PMCMC) methods. It compiles to C++, and supports multicore CPUs with SIMD instructions, GPUs and distributed memory clusters.

- **LibBi 2.x** attempts to incorporate the preceding considerations into a more general-purpose data-parallel probabilistic programming language. It still compiles to C++, and will eventually support the same hardware as LibBi 1.x.

- See www.libbi.org for LibBi 1.x. More information on LibBi 2.x, including an early release, will be available there soon.
Programming languages and platforms

- **General purpose**
  - C, C++, Java, Python...

- **Statistical/scientific computing**
  - MATLAB, Octave, R, Julia...

- **Symbolic manipulation**
  - Mathematica, Maxima, Maple...

- **Model specification**
  - BUGS, JAGS, Stan, Biips, LibBi 1.x...

- **Probabilistic programming languages**
  - Church, Venture, Anglican, LibBi 2.x...

- **Method specification**
  - NIMBLE...
Example #1: Simulating Gaussians

```plaintext
/**
 * Example program to simulate standard normal variates.
 *
 * `n`     Number of samples.
 * `mu`    Mean.
 * `sigma` Standard deviation.
 * `s`     Pseudorandom number seed.
 */

program example_gaussian(n:Integer ? 5, mu:Real ? 0.0, sigma:Real ? 1.0, 
                          s:Integer ? 0) {
  rng:RNG;
  seed(rng, s);

  x:Real;
  i:Integer <- 0;
  while (i < n) {
    simulate(rng, x ~ gaussian(x | mu, sigma));
    printf("%f ", x);
    i <- i + 1;
  }
  printf("\n");
}

> libbi example_gaussian -n 5 -s 10
  0.112526  1.158317  -0.722317  -0.014193  -1.323151
```
Example #1: Simulating Gaussians

```cpp
template<class RngType>
void simulate(RngType& rng, Real& x, Real& mu, Real& sigma)
{
  x = rng.gaussian<Real>(mu, sigma);
}
```

```cpp
gaussian<Real>(x:Real | mu:Real, sigma:Real)
```

---

Lawrence Murray
University of Oxford
Statistical Computing in the Big Data Context (and something about LibBi 2.x)
Example #2: Conjugate prior

/**
 * Example program to simulate posterior normal variates.
 *
 * `n`      Number of samples.
 * `mu0`    Prior mean on `mu`.
 * `sigma0` Prior standard deviation on `mu`.
 * `sigma`  Standard deviation.
 * `y`      Observation.
 * `s`      Pseudorandom number seed.
 */

program example_conjugate(n:Integer ? 5, mu0:Real ? 0.0, sigma0:Real ? 1.0, sigma:Real ? 1.0, y:Real ? 2.0, s:Integer ? 0) {
  rng:RNG;
  seed(rng, s);

  x:Real;
  i:Integer <- 0;
  while (i < n) {
    simulate(rng, x ~ gaussian(x | mu0, sigma0)*gaussian(y | x, sigma));
    printf("%f ", x);
    i <- i + 1;
  }
  printf("\n");
}

> libbi example_conjugate –n 5 –s 10 –y 2
1.079568 1.819054 0.489244 0.989964 0.064391
Example #2: Conjugate prior

```plaintext
    \[
    \text{simulate}(\text{rng}, x \sim \text{gaussian}(x \mid \mu_0, \sigma_0) \ast \text{gaussian}(y \mid x, \sigma))
    \]

/**
 * Simulate posterior distribution over unknown mean of Gaussian
 * with conjugate Gaussian prior.
 */

function simulate(rng:RNG, x:Real ~ gaussian(x \mid \mu_0:Real, \sigma_0:Real) *
    gaussian(y:Real \mid x, \sigma:Real)) {
    lambda0:Real <- 1.0/\sigma_0**2;
    lambda:Real <- 1.0/\sigma**2;

    mu1:Real <- (lambda0*\mu_0 + lambda*y)/(lambda0 + lambda);
    sigma1:Real <- sqrt(1.0/(lambda0 + lambda));

    simulate(rng, x \sim \text{gaussian}(x \mid \mu_1, \sigma_1));
}
```
Example #3: Indicator function

```plaintext
/**
 * Indicator function.
 *
 * `lower` Lower bound.
 * `upper` Upper bound.
 */
function indicator(x:Real | lower:Real, upper:Real);

/**
 * Simulate an expression that includes the indicator function.
 */
function simulate(rng:RNG, x:Real ~ @expr:*
    indicator(x | lower:Real, upper:Real)) {
    simulate(rng, x:Real ~ @expr);
    while (x < lower || x >= upper) {
        simulate(rng, x:Real ~ @expr);
    }
}
```
Example #3: Indicator function

```plaintext
/**
 * Indicator function.
 * `lower` Lower bound.
 * `upper` Upper bound.
 */
function indicator(x:Real | lower:Real, upper:Real);

/**
 * Simulate an expression that includes the indicator function.
 */
function simulate(rng:RNG, x:Real ~ @expr:
    indicator(x | lower:Real, upper:Real)) {
  simulate(rng, x:Real ~ @expr);
  while (x < lower || x >= upper) {
    simulate(rng, x:Real ~ @expr);
  }
}

/**
 * Simulate a truncated Gaussian.
 */
function simulate(rng:RNG, x:Real ~ gaussian(x | mu:Real, sigma:Real)*
    indicator(x | lower:Real, upper:Real)) {
  ...
}
```

Lawrence Murray
University of Oxford
Statistical Computing in the Big Data Context (and something about LibBi 2.x)
Expression matching

\[ x: \text{Real} \sim @\text{expr}: \]

\[ x: \text{Real} \sim \text{gaussian}(x \mid \mu: \text{Real}, \sigma: \text{Real}) \]

- Partial order on all code fragments.
- Method complexity can be handled using function overloads.
Example #4: Models

```cpp
/**
 * Generic model.
 */
model Model {
  function prior();

  function simulate_prior(rng:RNG) {
    simulate(rng, prior());
  }
}

/**
 * Specific model.
 */
model MyModel extends Model {
  variable mu:Real;
  variable sigma:Real;

  function prior() {
    mu ~ gaussian(mu | 0.0, 1.0);
    sigma ~ uniform(sigma | 0.0, 10.0);
  }
}
```
Example #4: Models

```plaintext
/**
 * Example program to simulate from a more complex model.
 *
 * `n` Number of samples.
 * `s` Pseudorandom number seed.
 * `output-file` Output file name.
 */

program example_model(n:Integer ? 5, s:Integer ? 0,
                      output_file:String ? "example_model.nc") {
    rng:RNG;
    seed(rng, s);

    m:MyModel[n];
    i:Integer <- 0;
    while (i < n) {
        m[i].simulate_prior();
        i <- i + 1;
    }

    out:NetCDFFile;
    out.open(output_file);
    create(out, m);
    write(out, m);
    close(out);
}
```

Lawrence Murray
University of Oxford

Statistical Computing in the Big Data Context (and something about LibBi 2.x)
Memory layout

```c
m:MyModel[n];
```

```
struct MyModel {
    double mu;
    double sigma;
}
MyModel m[n];
```

```
struct MyModel {
    double mu[n];
    double sigma[n];
}
MyModel m;
```

- "Array of structs" (left) type syntax is used, but a "struct of arrays" (right) implementation is used to facilitate data parallelism.
- Model complexity can be handled as in object-oriented languages, performance can be handled as in lower-level languages.

Lawrence Murray
University of Oxford

Statistical Computing in the Big Data Context (and something about LibBi 2.x)
Schema of models preserved in NetCDF files.

NetCDF is a high-performance binary format based on HDF5.
> ncdump example_model.nc

```c
netcdf example_model {
    dimensions:
        n = 10 ;
    group: m {
        dimensions:
            n = 10 ;
        variables:
            double mu(n) ;
            double sigma(n) ;
        data:
            mu = -0.917787219374395, -0.897920745559999...
            sigma = 7.15189364971593, 8.57945619849488...
    }
}
```
Summary

LibBi 2.x:

- Object-oriented paradigm for assembling complex models.
- Rich functions for assembling complex methods.
- Memory layouts facilitate data parallelism.
- More available soon at www.libbi.org