Today’s talk is about programming models
Today’s talk is about programming models for Bayesian inference and for scientific computing in general.
Today’s talk is about programming models for Bayesian inference and for scientific computing in general.

Existing approaches are trade-offs between ease-of-use + programmability flexibility + performance.
Today’s talk is about programming models for Bayesian inference and for scientific computing in general.

Existing approaches are trade-offs between ease-of-use + programmability flexibility + performance

Dedicated languages (e.g., JAGS)
- easy-to-use
- efficient for classic problems
- not well-suited to unusual problems and optimizations

Writing everything by hand in a low-level language like C++
- time-consuming
- difficult
- best performance (if done right)
- maximum flexibility
Large scale Bayesian inference
\( \sim 200,000 \) hours per run
\( \Leftrightarrow \) that’s 22 years!

Got 5.5M hours on the OCCIGEN supercomputer

We need
- parallel code
  \( \Leftrightarrow \) that takes advantage of the 50,000 cores of OCCIGEN
- high performance
- problem-specific optimizations
  \( \Leftrightarrow \) e.g., phylogeny-specific
- several versions

That’s a real software development challenge!

We need both flexibility + performance and a way to alleviate complexity
Software engineering could be like other industries and reuse pre-made components.

**Software components**: pieces of code that follow conventions to be interoperable with other components.

Components can be combined to build applications.

This approach is known to have good software engineering properties.

First part
Step-by-step example of component-based Metropolis-Hastings application.
Plan of the talk

First part
Step-by-step example of component-based Metropolis-Hastings application.

Second part
Presentation and results of
- tinycompo
  - our component model
- compoGM
  - our Bayesian inference library
A Simple Probabilistic Model

$$\alpha \sim \text{Exp}(1)$$

$$\lambda_i \sim \text{Gamma}(\alpha, \alpha)$$

$$K_{i,j} \sim \text{Poisson}(\lambda_i)$$

where

$$i \in \text{individuals}$$

$$j \in \text{experiments}$$
A Simple Probabilistic Model

\[ \alpha \sim \text{Exp}(1) \]
\[ \lambda_i \sim \text{Gamma}(\alpha, \alpha) \]
\[ K_{i,j} \sim \text{Poisson}(\lambda_i) \]

where

\[ i \in \text{individuals} \]
\[ j \in \text{experiments} \]
Our First Building Block

We are going to represent every probabilistic node with a structure called a component.

Every such component
- needs to access the value of its parent \( p \)
- is associated to a distribution \( D \)
- can give its value \( x \)
- can give its “log prob” \( \log(f_D(x; p)) \)
Our First Building Block

We are going to represent every probabilistic node with a structure called a component.

Every such component:
- needs to access the value of its parent $p$
- is associated to a distribution $D$
- can give its value $x$
- can give its “log prob” $\log(f_D(x; p))$

Legend:
- $T$ component of type $T$
- $T$ the component provides functionality $T$
- $T$ the component needs functionality $T$ to work
- ‘name’ the name of something
- $T<U>$ type $T$ with subtype $U$
The cyan bits are called **ports**

Ports are used by components to interact with other components

Everything except ports is hidden inside components

Components are **black boxes**
component 'lambda'
  of type ProbNode<Gamma>
  with params 1, 1

component 'K'
  of type ProbNode<Poisson>
  connect 'p' to 'lambda'
    using Use<Value>

Vincent Lanore (CNRS)  Bayesian Inference with Components  June 21st, 2018  9 / 30
component 'lambda'
of type ProbNode<Gamma>
with params 1, 1

component 'K'
of type ProbNode<Poisson>
connect 'p' to 'lambda'
using Use<Value>

Legend
initialize component with parameters p...
connect these ports using connector T
The most basic connection is one port using a functionality provided by another port.

\[ \text{Use} \quad \text{<Value>} \quad \text{execution} \quad \text{Value} \]
Connectors

The most basic connection is one port using a functionality provided by another port

In the general case, connectors are functions that decide how to connect components
Connectors

The most basic connection is one port using a functionality provided by another port:

\[ \text{Use } \langle \text{Value} \rangle \rightarrow \text{Value} \]

In the general case, connectors are functions that decide how to connect components:

\[ \text{ManyToOne } \langle \text{C} \rangle \rightarrow \text{C} \]

\[ \text{ManyToOne } \langle \text{Use } \langle \text{Value} \rangle \rangle \rightarrow \text{Value } \text{Value} \]
We need some way to declare arrays of components
We need some way to declare arrays of components

Legend

{0, 1, 2}

composite with
its contents

Array

$K$

<ProbNode

<Poisson>

>

execution

ProbNode

<Poisson>

Value

LogProb

'p'

ProbNode

<Poisson>

Value

LogProb

'p'

ProbNode

<Poisson>

Value

LogProb

'p'

ProbNode

<Poisson>

Value

LogProb

'p'

<K>

0

1

2

...
Composites are collections of components that act as components
Laying Out the Basic Data Structure

\[ \alpha \sim \text{Exp}(1) \]
\[ \lambda_i \sim \text{Gamma}(\alpha, \alpha) \]
\[ K_{i,j} \sim \text{Poisson}(\lambda_i) \]
Laying Out the Basic Data Structure

\textbf{component 'alpha'}
\textit{of type ProbNode<Exp>}
\textit{with params 1}

\( \alpha \sim \text{Exp}(1) \)
\( \lambda_i \sim \text{Gamma}(\alpha, \alpha) \)
\( K_{i,j} \sim \text{Poisson}(\lambda_i) \)
Laying Out the Basic Data Structure

\[ \alpha \sim \text{Exp}(1) \]
\[ \lambda_i \sim \text{Gamma}(\alpha, \alpha) \]
\[ K_{i,j} \sim \text{Poisson}(\lambda_i) \]

*component 'alpha'*
  - of type ProbNode<Exp>
  - with params 1

*component 'lambda'*
  - of type Array<ProbNode<Gamma>>
  - with params individuals
  - connect 'p' to 'alpha'
    - using ManyToOne<Use<Value>>
Laying Out the Basic Data Structure

\[ \alpha \sim \text{Exp}(1) \]
\[ \lambda_i \sim \text{Gamma}(\alpha, \alpha) \]
\[ K_{i,j} \sim \text{Poisson}(\lambda_i) \]

component 'alpha'
of type ProbNode<Exp>
with params 1

component 'lambda'
of type Array<ProbNode<Gamma>>
with params individuals
connect 'p' to 'alpha'
    using ManyToOne<Use<Value>>

component 'K'
of type Matrix<ProbNode<Exp>>
with params individuals, experiments
connect 'p' to 'lambda'
    using ManyToMany<ManyToOne<Use<Value>>>

Laying Out the Basic Data Structure

component 'alpha'
of type ProbNode<Exp>
with params 1

component 'lambda'
of type Array<ProbNode<Gamma>>
with params individuals
connect 'p' to 'alpha'
using ManyToOne<Use<Value>>

component 'K'
of type Matrix<ProbNode<Exp>>
with params individuals, experiments
connect 'p' to 'lambda'
using ManyToMany<ManyToOne<Use<Value>>>
What We Have So Far

- 'alpha'
  - ProbNode <Exp>
    - 1
- 'p'
  - ManyToOne<User<Value>>
    - ManyToMany<ManyToOne<
      Use<Value>>>
      - individuals
      - 'p'
  - Array 'lambda'
    - <ProbNode <Gamma>
      >
      - individuals
- 'p'
  - ManyToMany<ManyToOne<
      Use<Value>>>
    - Matrix 'K'
      - <ProbNode <Poisson>
        >
        - individuals, experiments
What We Have So Far

Introduced several concepts

- **components**
  - basic building block
- **ports**
  - interaction point w/ other components
- **connectors**
  - functions that decide how to connect
- **composites**
  - component collections that act as components
What We Have So Far

Introduced several concepts
- **components**
  - basic building block
- **ports**
  - interaction point w/ other components
- **connectors**
  - functions that decide how to connect
- **composites**
  - component collections that act as components

Probabilistic model data structure
- access value of nodes
- get likelihood of nodes
Metropolis-Hastings Moves

**MH algorithm**

- given parameter vector $\theta$
- propose a change $\theta'$ according to proposal distribution $q$
- accept change with probability (reject otherwise)

$$
\min \left( \frac{\pi(\theta')q(\theta'|\theta')}{\pi(\theta)q(\theta'|\theta)}, 1 \right)
$$

where $\pi$ is the distribution of interest

- start over
MH algorithm

- given parameter vector $\theta$
- propose a change $\theta'$ according to proposal distribution $q$
- accept change with probability (reject otherwise)

$$\min \left( \frac{\pi(\theta') q(\theta|\theta')}{\pi(\theta) q(\theta'|\theta)}, 1 \right)$$

where $\pi$ is the distribution of interest

- start over

In practice, we compute

$$\log\left( \frac{\pi(\theta')}{\pi(\theta)} \right)$$

$$= \sum_{n \in \text{nodes}} \log(f_{D_n}(p'_n)) - \log(f_{D_n}(p_n))$$

$$= \sum_{n \in \text{nodes'}} \log(f_{D_n}(p'_n)) - \log(f_{D_n}(p_n))$$

where $\text{nodes'}$ is the set of nodes whose logprob is changed by the proposed move
MHMove Component

MHMove component

- needs access to a target’s value
- needs access to the logprob of all nodes whose logprob it might affect
- is associated to proposal distribution $M$
- provides a “go” port which performs a move
Adding Moves to Our Assembly

component 'move_alpha'
of type MHMove<Scale>
connect 'target' to 'alpha'
  using Use<Value>
connect 'logprobs' to 'alpha'
  using Use<LogProb>
connect 'logprobs' to 'lambda'
  using OneToMany<Use<LogProb>>
It would be nice to auto-compute the list of nodes a move needs to be connected to

(the so-called Markov blanket)

that’s the job of a new connector!
It would be nice to auto-compute the list of nodes a move needs to be connected to

(the so-called Markov blanket)

that’s the job of a new connector!
Adding Moves to Our Assembly

component 'move_alpha' of type MHMove<Scale>
connect 'target', 'logprobs' to 'alpha', 'model'
using ConnectMove
More Moves

- `model`:
  - `p`:
    - `Poisson`
      - `k`:
        - `Gamma`
          - `lambda`:
            - `p`:
              - `Exp`
                - `alpha`

- `ConnectMove`:
  - `move_alpha`:
    - `logprobs`

- `MHMove`:
  - `move_lambda`:
    - `logprobs`

- `Array`:
  - `Matrix`:
    - `K`:
      - `Poisson`

- `ManyToMany<ManyToOne<Use<Value>>>`
  - `individuals`, `experiments`

- `Go`
  - `target`
  - `ClampData`
    - `data`
**Full Program**

```c
main () {
    /* model declaration */
    model = {
        ...
    }

    /* move declarations */
    moves = {
        ...
    }

    /* iteration loop */
    for iteration in {1,... N}
        for move in moves
            move .go ()
            write_trace ( model , 'trace.tsv')
}
```

So far, we have
- built a probabilistic model data structure
- added Metropolis-Hastings moves
- a simple main that runs the iteration loop
When performing a MH move on $\alpha$ we must compute
\[
\log(\pi(\theta)) = \log(f_{\text{Exp}}(\alpha; 1)) + \sum_i \log(f_{\text{Gamma}}(\lambda_i; \alpha, \alpha)) + \sum_{i,j} \log(f_{\text{Poisson}}(K_{i,j}; \lambda_i))
\]
Sufficient Statistics

\( \alpha \sim \text{Exp}(1) \)

\( \lambda_i \sim \text{Gamma}(\alpha, \alpha) \)

\( K_{i,j} \sim \text{Poisson}(\lambda_i) \)

When performing a MH move on \( \alpha \) we must compute

\[
\log(\pi(\theta)) = \log(f_{\text{Exp}}(\alpha; 1)) + \sum_i \log(f_{\text{Gamma}}(\lambda_i; \alpha, \alpha)) + \sum_{i,j} \log(f_{\text{Poisson}}(K_{i,j}; \lambda_i))
\]

We can rewrite the red part

\[
\sum_i \log(f_{\text{Gamma}}(\lambda_i; \alpha, \alpha)) = \sum_i \log\left(\frac{\alpha^\alpha}{\Gamma(\alpha)} \lambda_i^{\alpha-1} e^{-\alpha \lambda_i}\right) = N\alpha \log(\alpha) - N \log(\Gamma(\alpha)) + (\alpha - 1) \sum_i \log(\lambda_i) - \alpha \sum_i \lambda_i
\]

Blue parts don’t depend on \( \alpha \) and can be pre-computed

These are sufficient statistics
**New type of component!**

**GammaSuffStat** component

- needs access to an array of gamma node values
- can be told to gather the sufficient statistics, i.e., compute $\sum_i \lambda_i$ and $\sum_i \log(\lambda_i)$
- can be told that the statistics are no longer valid (corrupted)
- can give the log prob of the whole array
New Assembly

Vincent Lanore (CNRS)
main() {
    model = { ... }
    move_alpha = { ... }
    suffstats = { ... }
    moves_lambda = { ... }

    /* iteration loop */
    for iteration in {1,... N}
        for move in moves_lambda
            move.go()

    /* gather and move */
    suffstats.gather()
    for rep in {1,... 10}
        move_alpha.go()
    suffstats.corrupt()

    write_trace(model, 'trace.tsv')
}

In the end, we have
- built a probabilistic model
- data structure
- added Metropolis-Hastings moves
- a simple main that runs the iteration loop
- optimized one move using sufficient statistics
We have implemented

- tinycompo, a C++ generic component model implementation
  ↩️ it’s on github: https://github.com/vlanore/tinycompo

- compoGM, a tinycompo-based Bayesian inference library
  ↩️ it’s on github: https://github.com/vlanore/compoGM

Today's example can be implemented with compoGM (see src/M0.cpp on the github)

And more!
- multi-threaded versions
- distributed (MPI) versions

Both codes are well-tested and functional research prototypes
model = {
    component 'lambda'
        of type ProbNode<Gamma>
        with params 1, 1
    component 'K'
        of type ProbNode<Poisson>
        connect 'p' to 'lambda'
            using Use<Value>
}
class MyLog: public tc::Component, public Value<double> {
    Value<double>* target;

    public:
        MyLog() { port("target", &MyLog::target); }

        double& get_ref() final const { return log(target->get_ref()); }
Comparison with RevBayes and JAGS

We compared compoGM with JAGS and RevBayes using 3 models taken from a bioinformatics use-case

<table>
<thead>
<tr>
<th></th>
<th>Lines of Code</th>
<th>Time</th>
<th>ESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Today’s example</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 1, compoGM</td>
<td>77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 1, RevBayes</td>
<td>65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 1, JAGS</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 2, compoGM</td>
<td>110</td>
<td>2m19s</td>
<td>2021</td>
</tr>
<tr>
<td>Model 2, RevBayes</td>
<td>50</td>
<td>36min25s?</td>
<td>4609</td>
</tr>
<tr>
<td>Model 2, JAGS</td>
<td>55</td>
<td>1m15s</td>
<td>1537</td>
</tr>
<tr>
<td>Model 3, compoGM</td>
<td>148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 3, JAGS</td>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thanks to Philippe Veber for JAGS scripts and to Bastien Boussau for performance measurement and RevBayes scripts

- Lines of code computed using cloc
- Time for 5,000 iterations. Iteration meaning dependent on program.
- ESS is mean Effective Sample Size for a subset of probabilistic nodes.
Conclusion

Problem
Design a Bayesian inference code while reconciling ease-of-use, flexibility and performance

Proposed solution
Use a component-based approach

Today
- illustrated component-based approach on a simple example
- presented tinycompo and compoGM, our C++ implementations

Perspectives
- improve performance further
- better MPI and thread support
- convergence detection using compoGM
Conclusion

Problem
Design a Bayesian inference code while reconciling ease-of-use, flexibility and performance

Proposed solution
Use a component-based approach

Today
- illustrated component-based approach on a simple example
- presented tinycompo and compoGM, our C++ implementations

Perspectives
- improve performance further
- better MPI and thread support
- convergence detection using compoGM

Thank you for your attention!