Building High-performance Bayesian Inference Applications with Software Components

Vincent Lanore

Univ. Lyon, UCBL, CNRS
LBBE, UMR5558

vincent.lanore@univ-lyon1.fr

June 21st, 2018

Programming Models

Today's talk is about programming models

Programming Models

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

Programming Models

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

Programming Models

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

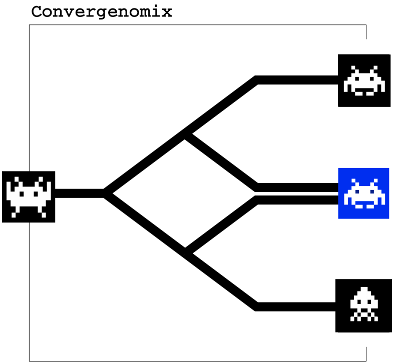
Dedicated languages
(e.g., JAGS)

- 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

A Little Bit of Context



Large scale Bayesian inference
$\sim$ 200 000 hours per run
↳ that's 22 years!

Got 5.5M hours on the OCCIGEN supercomputer

We need

- parallel code
  ↳ that takes advantage of the 50 000 cores of OCCIGEN
- high performance
- problem-specific optimizations
  ↳ e.g., phylogeny-specific
- several versions

That's a real software development challenge!

We need both flexibility + performance and a way to alleviate complexity

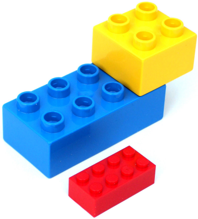
Software Components

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

USB 2.0

Components can be combined to build applications

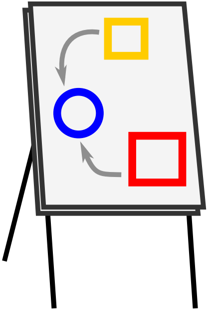


Source: Wikipedia

This approach is known to have good software engineering properties

Plan of the talk

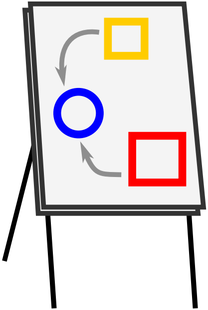
First part
Step-by-step example of component-based Metropolis-Hastings application.



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

$$\begin{aligned}
\alpha &\sim \mathit{Exp}(1)\\
\lambda_i &\sim \mathit{Gamma}(\alpha, \alpha)\\
K_{i,j} &\sim \mathit{Poisson}(\lambda_i)
\end{aligned}$$

where

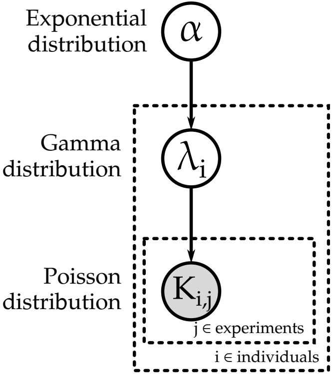
$$\begin{aligned}
i &\in \mathit{individuals}\\
j &\in \mathit{experiments}
\end{aligned}$$

A Simple Probabilistic Model

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

where

$$i \in individuals$$
$$j \in experiments$$



Our First Building Block

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

Every such component

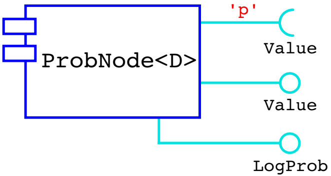
- needs to access the value of its parent p
- is associated to a distribution D
- can give its value x
- can give its "log prob" $log(f_D(x; p))$

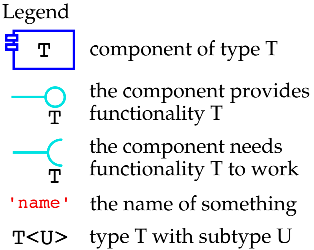
Our First Building Block

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

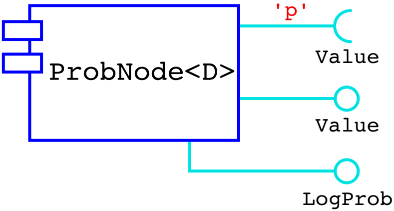
Every such component

- 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” $log(f_D(x; p))$





Ports



The cyan bits are called ports

Ports are used by components to interact with other components

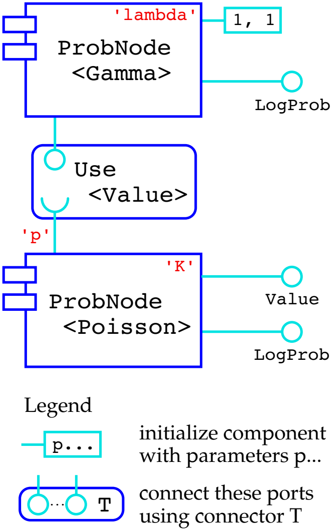
Everything except ports is hidden inside components
Components are black boxes

Connecting Components

```
component 'lambda'
  of type ProbNode<Gamma>
  with params 1, 1

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

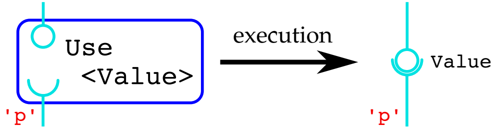
Connecting Components

```
component 'lambda'
  of type ProbNode<Gamma>
  with params 1, 1

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



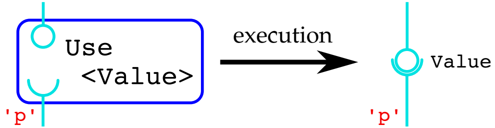
Connectors

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

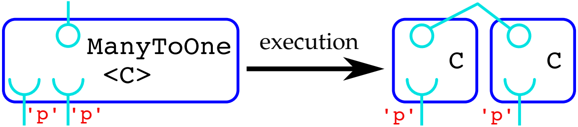


Connectors

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

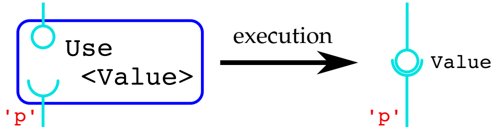


In the general case, connectors are functions that
decide how to connect components

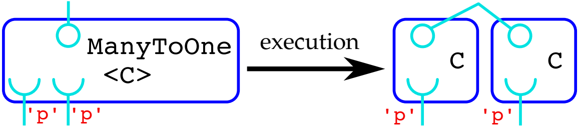


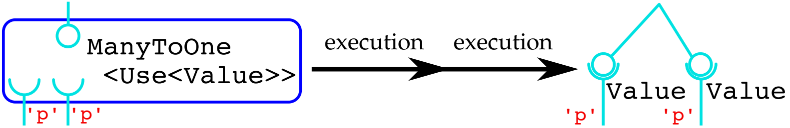
Connectors

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



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



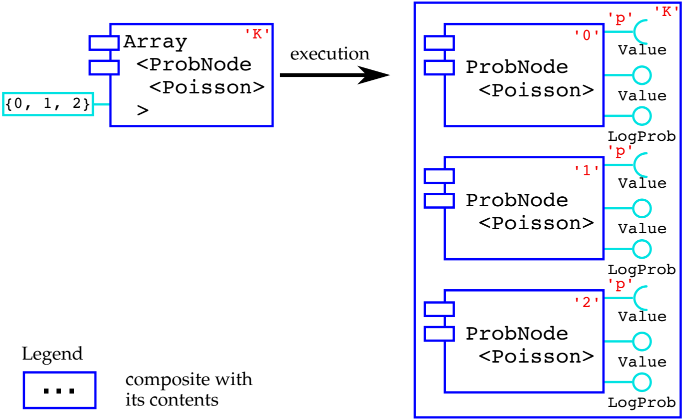


Composites

We need some way to declare arrays of components

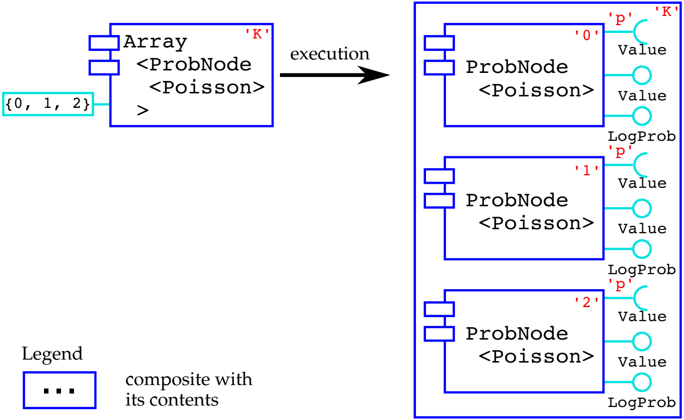
Composites

We need some way to declare arrays of components



Composites

We need some way to declare arrays of components



Composites are collections of components that act as components

Laying Out the Basic Data Structure

$$\begin{aligned} \alpha &\sim \mathit{Exp}(1) \\ \lambda_i &\sim \mathit{Gamma}(\alpha, \alpha) \\ K_{i,j} &\sim \mathit{Poisson}(\lambda_i) \end{aligned}$$

Laying Out the Basic Data Structure

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

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

Laying Out the Basic Data Structure

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

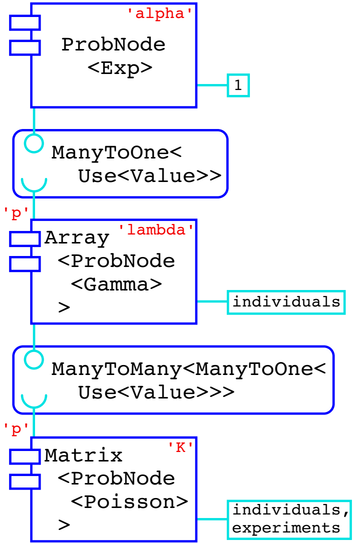
```
component 'alpha'
  of type ProbNode<Exp>
  with params 1

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

Laying Out the Basic Data Structure

$$\alpha \sim \mathit{Exp}(1)$$
$$\lambda_i \sim \mathit{Gamma}(\alpha, \alpha)$$
$$K_{i,j} \sim \mathit{Poisson}(\lambda_i)$$

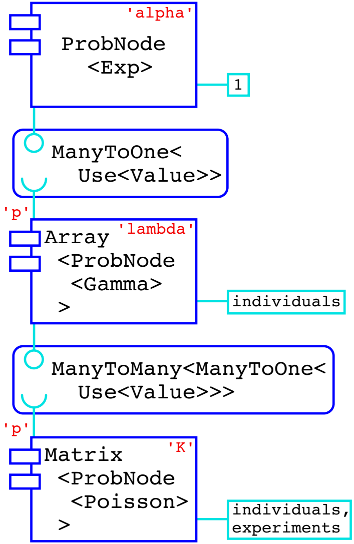
```
component 'alpha'
  of type ProbNode<Exp>
  with params 1

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

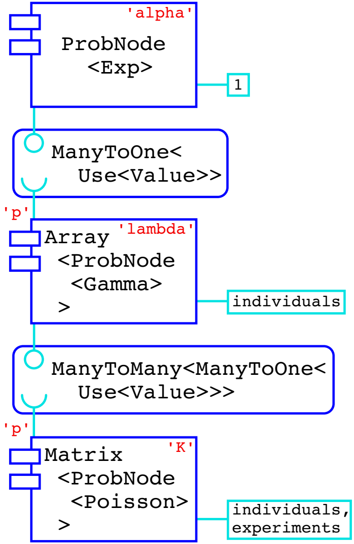



```
component 'alpha'
  of type ProbNode<Exp>
  with params 1

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



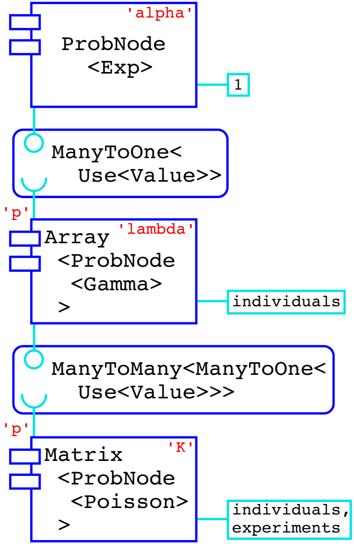
What We Have So Far



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 θ
- propose a change θ' 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 π is the distribution of interest

- start over

Metropolis-Hastings Moves

MH algorithm

- given parameter vector θ
- propose a change θ' 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 π is the distribution of interest

- start over

In practice, we compute

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

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

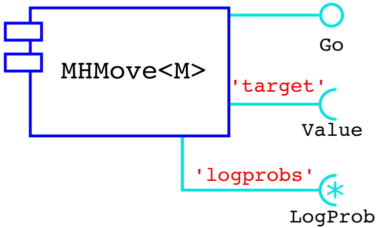
$$= \sum_{n \in nodes'} log(f_{D_n}(p'_n)) - log(f_{D_n}(p_n))$$

where *nodes′* is the set of nodes whose logprob is changed by the proposed move

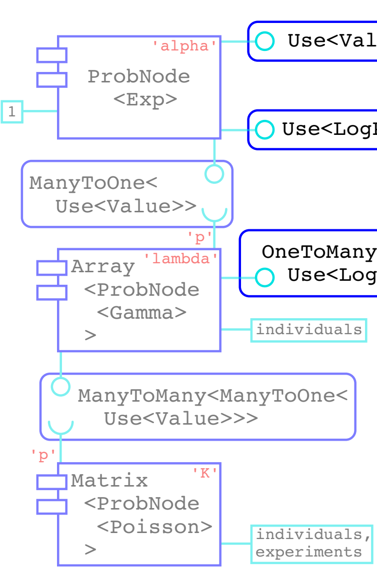
New type of component!

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>>
```

ConnectMove Connector

It would be nice to auto-compute the list of nodes a move needs to be connected to

(the so-called Markov blanket)

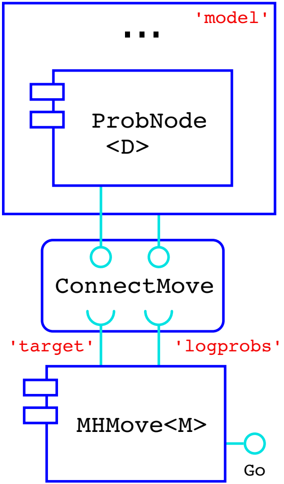
that's the job of a new connector!

ConnectMove Connector

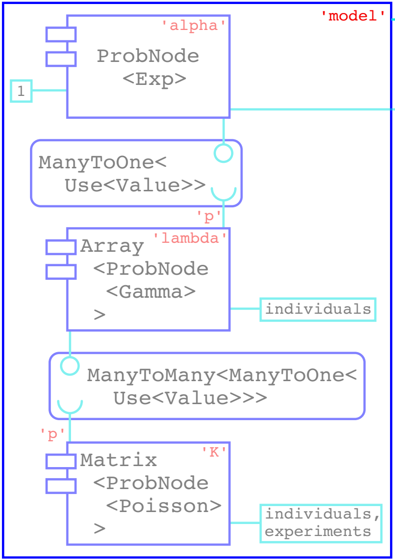
It would be nice to auto-compute the list of nodes a move needs to be connected to

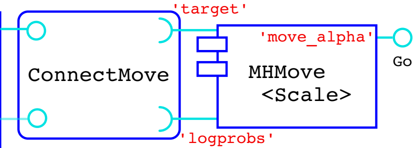
(the so-called Markov blanket)

that's the job of a new connector!



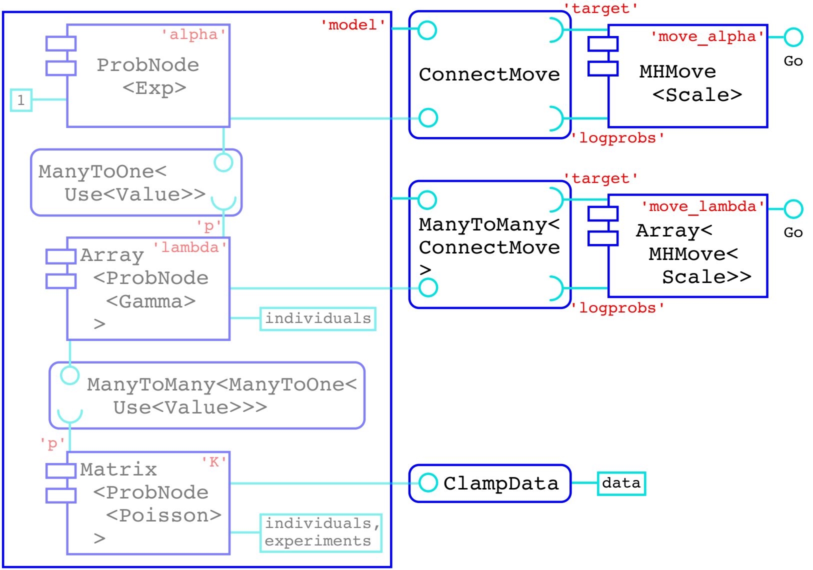
Adding Moves to Our Assembly






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

More Moves



'model'
'alpha'
ProbNode
<Exp>
1
ManyToOne<
Use<Value>>
'p'
Array
'lambda'
<ProbNode
<Gamma>
>
individuals
ManyToMany<ManyToOne<
Use<Value>>>
'p'
Matrix
'K'
<ProbNode
<Poisson>
>
individuals,
experiments
ConnectMove
'target'
'logprobs'
'move_alpha'
MHMove
<Scale>
Go
ManyToMany<
ConnectMove
>
'target'
'logprobs'
'move_lambda'
Array<
MHMove<
Scale>>
Go
ClampData
data

Full Program

```
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

Sufficient Statistics

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

When performing a MH move on α we must compute

$$\begin{aligned} &log(\pi(\theta)) \\ &= log(f_{Exp}(\alpha; 1)) \\ &\quad + \sum_i log(f_{Gamma}(\lambda_i; \alpha, \alpha)) \\ &\quad + \sum_{i,j} log(f_{Poisson}(K_{i,j}; \lambda_i)) \end{aligned}$$

Sufficient Statistics

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

When performing a MH move on α we must compute

$$\begin{aligned} &log(\pi(\theta)) \\ =& log(f_{Exp}(\alpha; 1)) \\ &+ \sum_i log(f_{Gamma}(\lambda_i; \alpha, \alpha)) \\ &+ \sum_{i,j} log(f_{Poisson}(K_{i,j}; \lambda_i)) \end{aligned}$$

We can rewrite the red part

$$\begin{aligned} &\sum_i log(f_{Gamma}(\lambda_i; \alpha, \alpha)) \\ =& \sum_i log(\frac{\alpha^\alpha}{\Gamma(\alpha)} \lambda_i^{\alpha-1} e^{-\alpha\lambda_i}) \\ =& N\alpha log(\alpha) - Nlog(\Gamma(\alpha)) \\ &+ (\alpha - 1)\sum_i log(\lambda_i) - \alpha \sum_i \lambda_i \end{aligned}$$

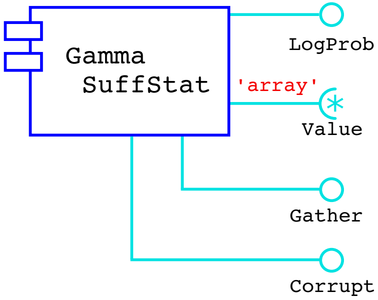
Blue parts don't depend on α and can be pre-computed

These are sufficient statistics

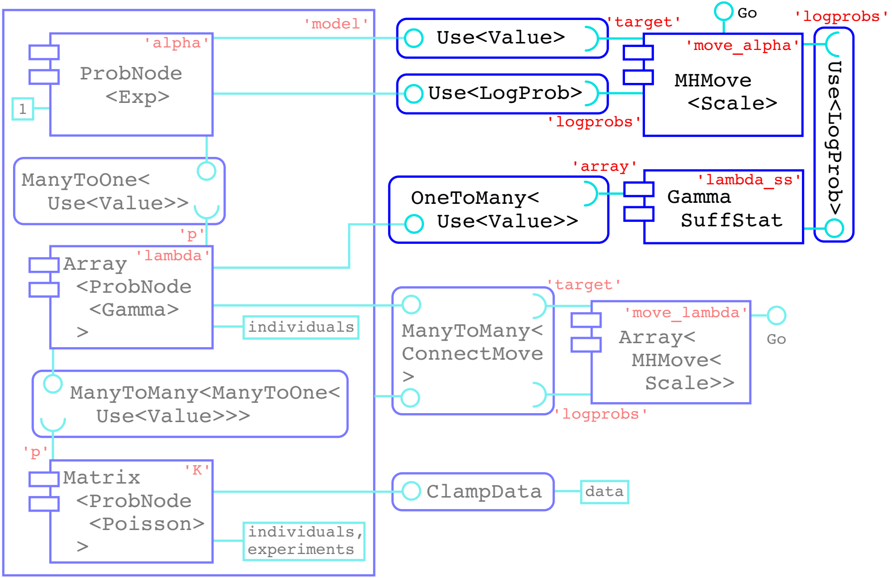
New type of component!

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



Full Program

```
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

tinycompo and compoGM

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 gihtub)

And more!

- multi-threaded versions
- distributed (MPI) versions

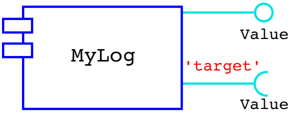
Both codes are well-tested and functional research prototypes

Comparison between Pseudocode and tinycompo

```
model = {

  component 'lambda'
    of type ProbNode<Gamma>
    with params 1, 1

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

}
```

```
tc::Model m;

m.component<OrphanNode<Gamma>>(
    "lambda", 1, 1);

m.component<UnaryNode<Poisson>>("K")
    .connect<Use<Value<double>>>(
      "p", "lambda");
```

Defining a Custom Component




```
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

	Lines of Code	Time	ESS
Today's example	64		
Model 1, compoGM	77		
Model 1, RevBayes	65		
Model 1, JAGS	48		
Model 2, compoGM	110	2m19s	2021
Model 2, RevBayes	50	36min25s?	4609
Model 2, JAGS	55	1m15s	1537
Model 3, compoGM	148		
Model 3, JAGS	60		

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!