Automated learning with a probabilistic programming language: Birch

Lawrence Murray

Department of Information Technology, Uppsala University

Outline

1. The Birch probabilistic programming language.
2. The delayed sampling heuristic.
3. A probabilistic definition of probabilistic programs.
4. Example: multiple object tracking.

UPPSALA UNIVERSITET

Swedish Foundation *for* Strategic Research

Birch (`birch-lang.org`)

- Object-oriented and probabilistic programming paradigms.
- Draws inspiration from many places, including LibBi, for which it is something of a successor, but also modern object-oriented languages such as Swift.
- Maintains the C/C++ basis of LibBi: compiles to C++14, uses standard C/C++ libraries for numerical computing such as Boost and Eigen.
- Multithreaded using OpenMP.
- Dynamic memory management, with reference counting.
- Free and open source, under the Apache 2.0 license.

Birch

- Preferably, models are implemented by defining the **joint distribution**.
- That is, program code does not distinguish between latent and observed variables, this distinction is made at runtime according to value assignment.
- Inference methods are also implemented in the Birch language.
- A particular feature of Birch is **delayed sampling**, a dynamic mechanism for full and partial analytical solutions.

Delayed sampling

- Automatically yields optimizations such as variable elimination, Rao–Blackwellization, and locally-optimal proposals.
- Is a **heuristic**: we have only partial knowledge of the whole model structure during execution and must make myopic decisions.
- Usually, as a probabilistic program executes, we eagerly sample each latent variable and update a **weight** for each observed variable.
- Instead, we delay the sampling of each latent variable in order to analytically **condition** on each observed variable where possible.
- In Birch, this is implemented via computational graphs and implicit type conversion.

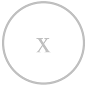
L. M. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. B. Schön. Delayed sampling and automatic Rao–Blackwellization of probabilistic programs. In Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS), Lanzarote, Spain, 2018. URL `arxiv.org/abs/1708.07787`

Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	

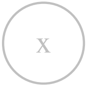
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	**assume** x
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



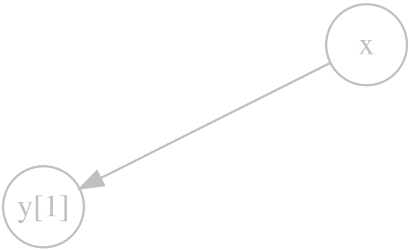
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



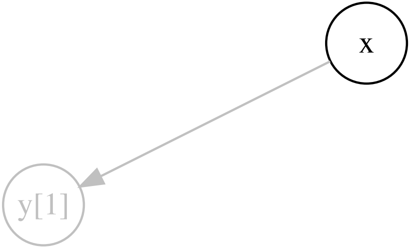
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



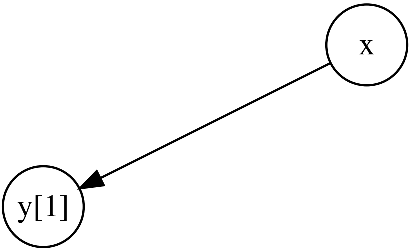
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



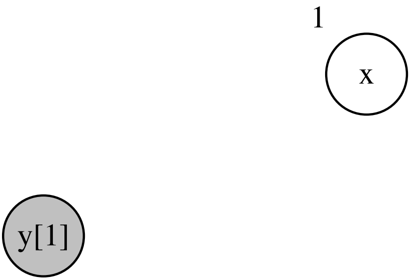
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



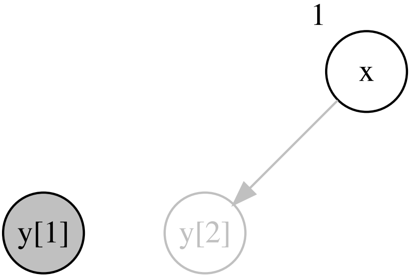
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



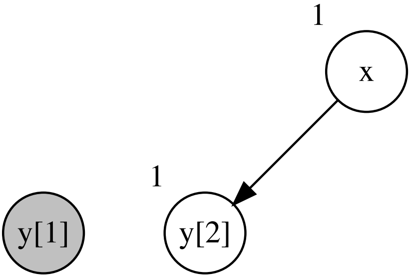
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



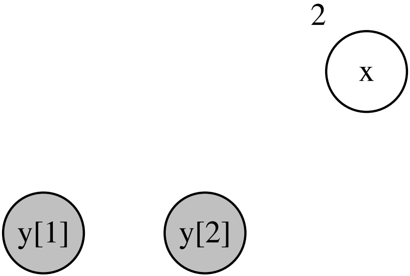
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



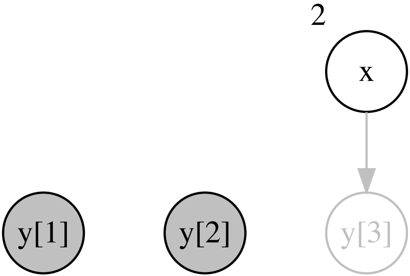
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



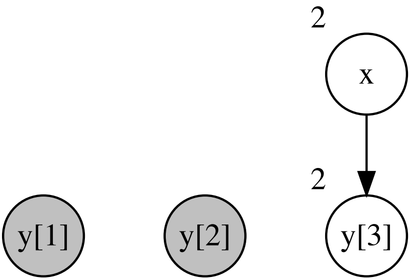
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



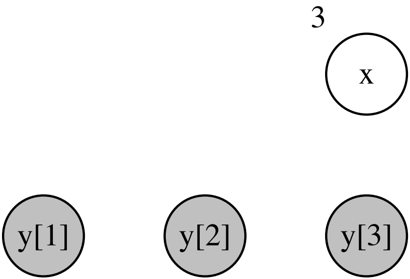
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



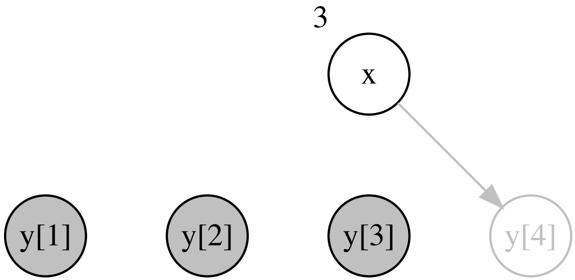
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



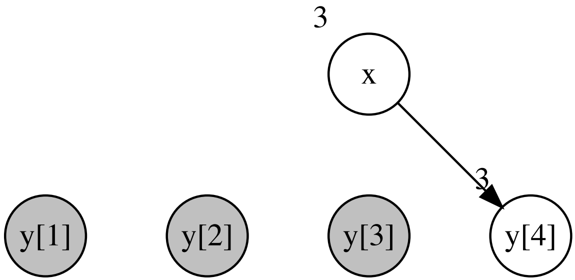
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



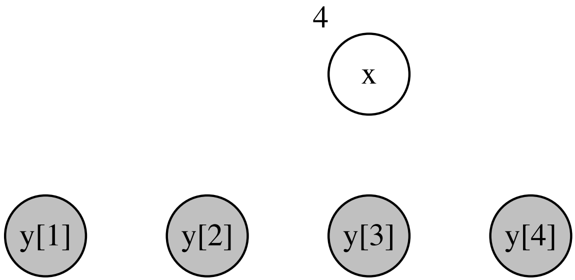
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



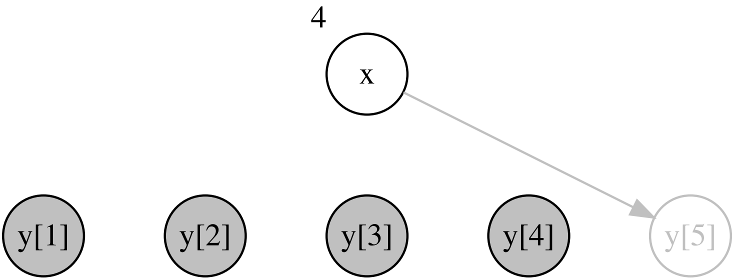
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



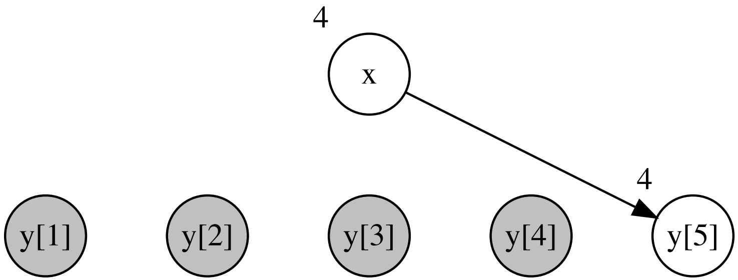
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



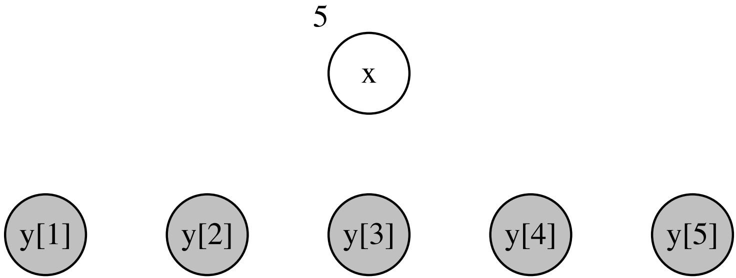
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



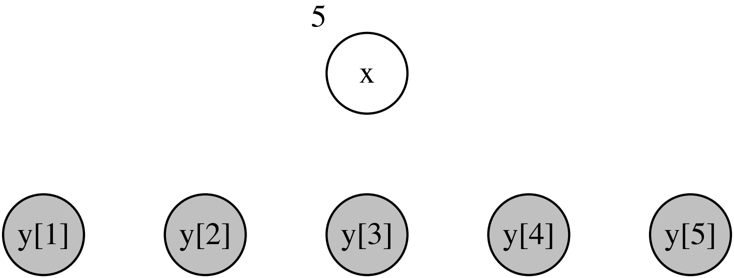
Example #1

Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	**observe** y[n]
`}`	
`stdout.print(x);`	



Example #1

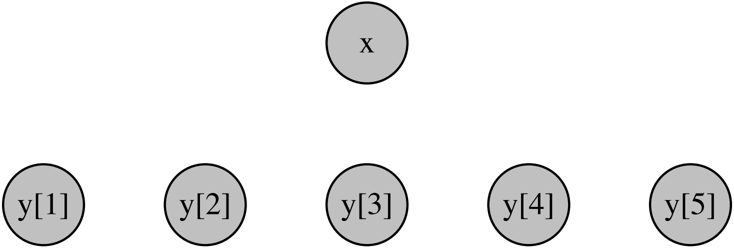
Code	Checkpoint
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	**value** x



Example #1

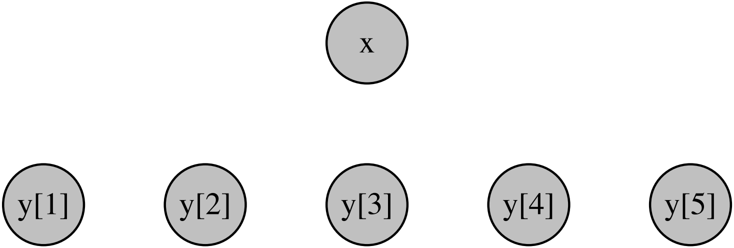
Code	Checkpoint

```
x ~ Gaussian(0.0, 1.0);
for (n in 1..N) {
  y[n] ~ Gaussian(x, 1.0);
}
stdout.print(x);
```



Example #1

Code	**Checkpoint**
`x ~ Gaussian(0.0, 1.0);`	
`for (n in 1..N) {`	
`  y[n] ~ Gaussian(x, 1.0);`	
`}`	
`stdout.print(x);`	



Example #2

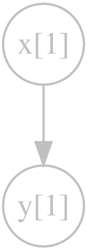
Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	

Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	**assume** x[1]
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	

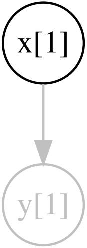
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



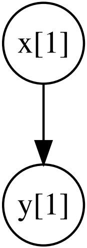
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



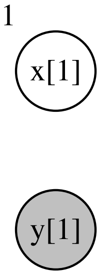
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	**observe** y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



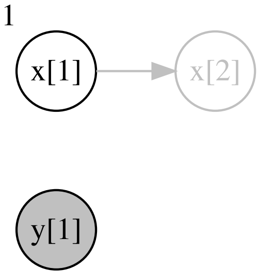
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	observe y[1]
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



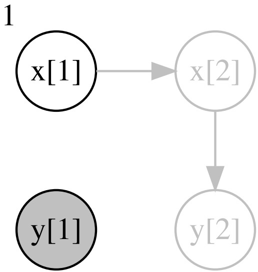
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



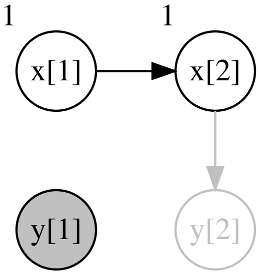
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



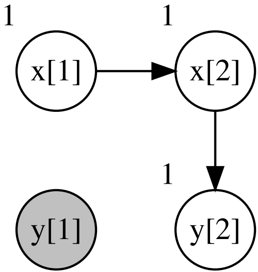
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



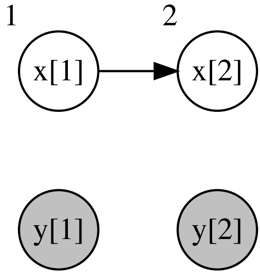
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



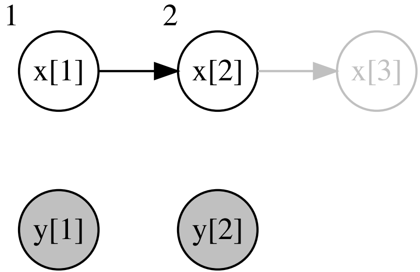
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



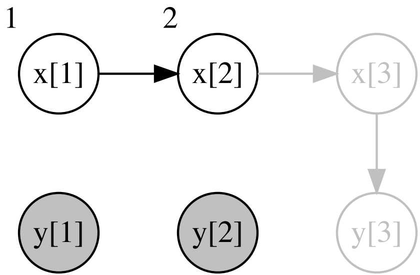
Example #2

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



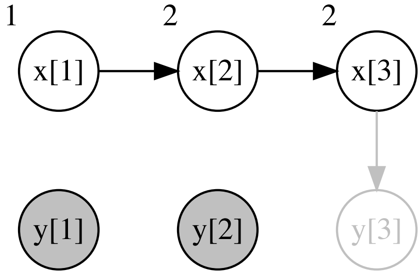
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



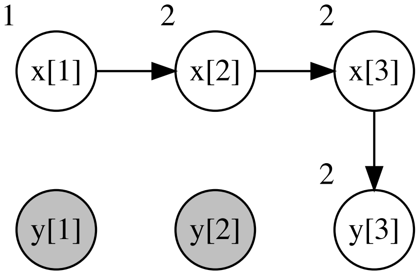
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



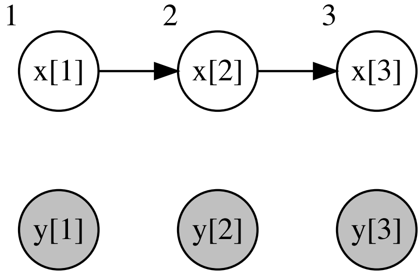
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



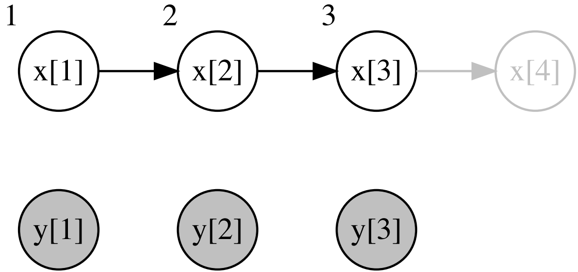
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



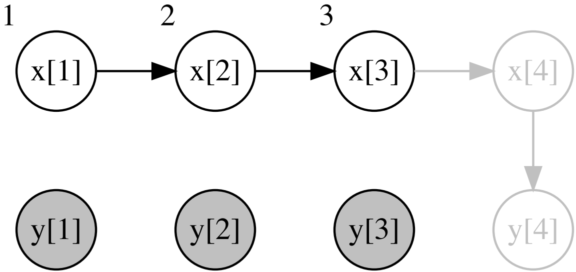
Example #2

Code	**Checkpoint**
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



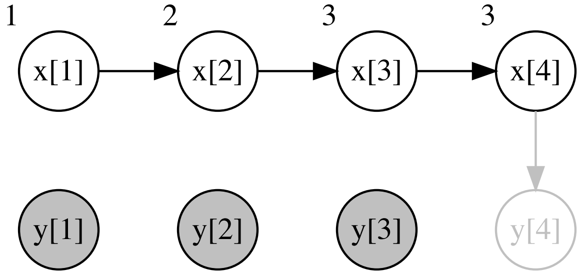
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



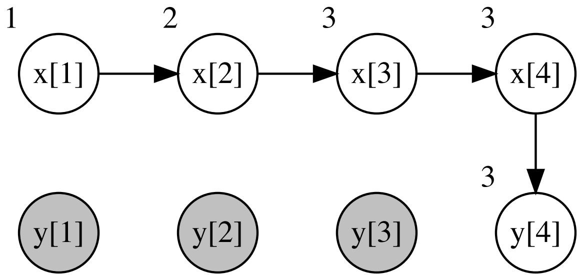
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



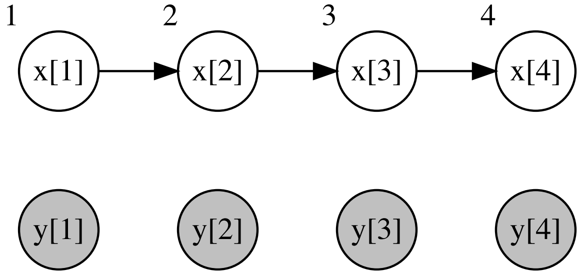
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



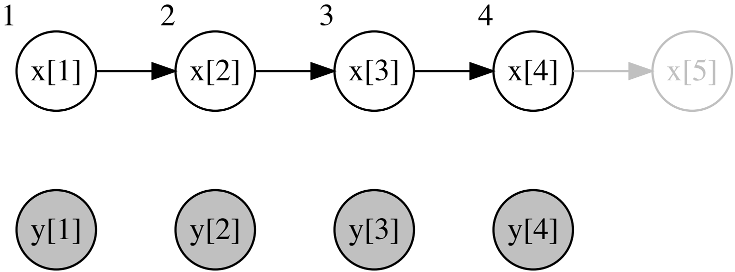
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



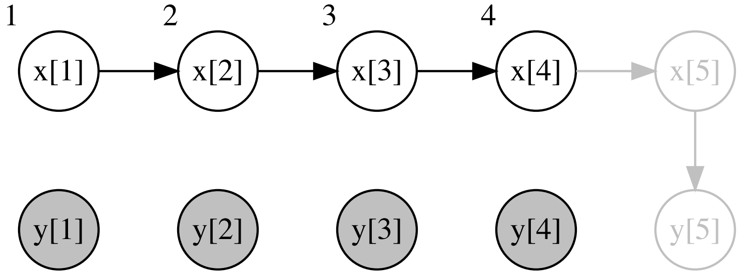
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	**assume** x[t]
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	



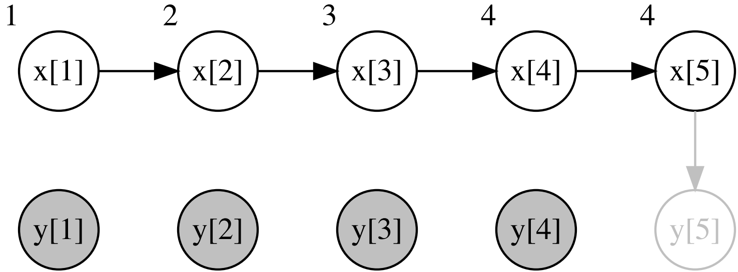
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



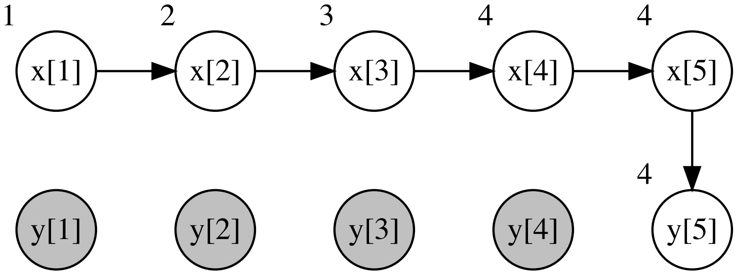
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



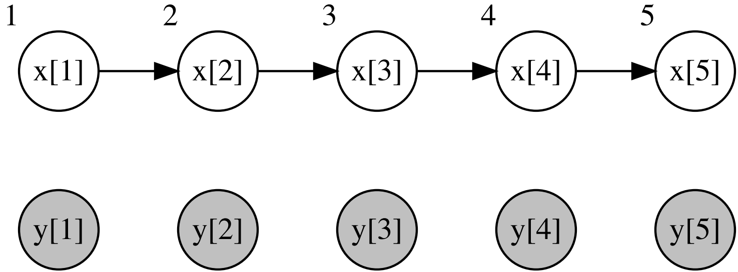
Example #2

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



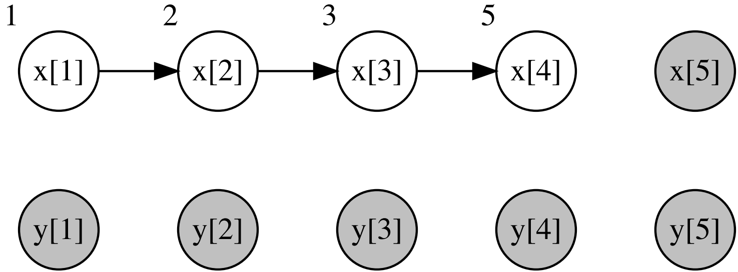
Example #2: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	**observe** y[t]
`}`	
`stdout.print(x[1]);`	



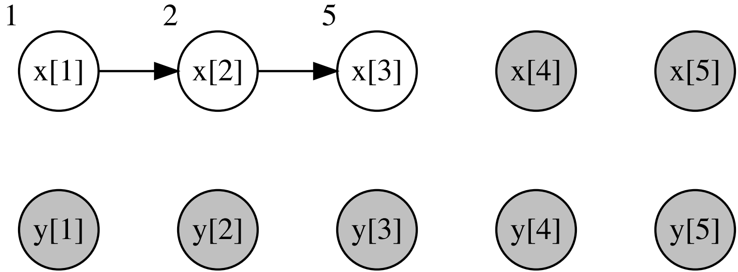
Example #2: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



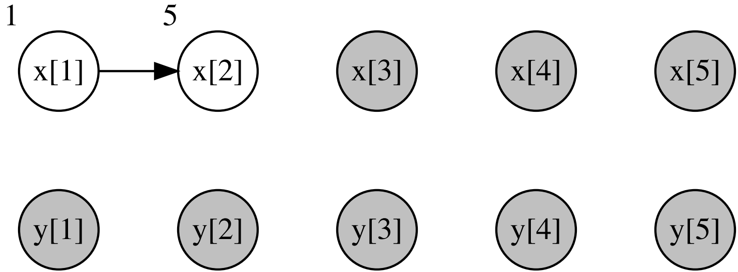
Example #2: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



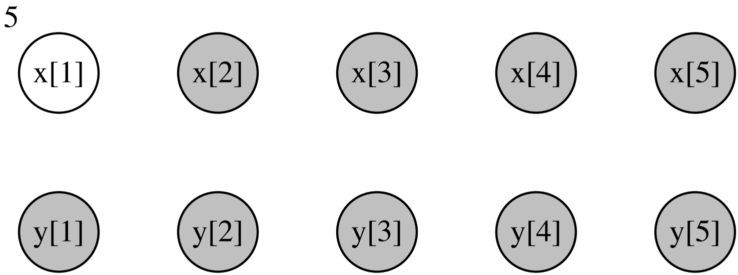
Example #2: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



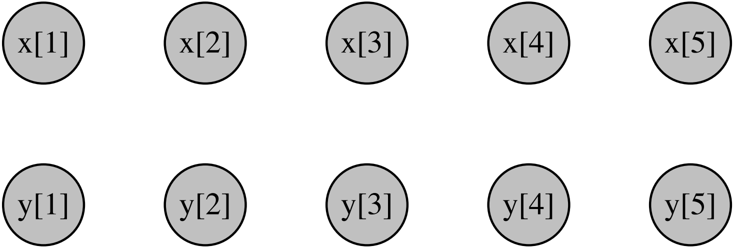
Example #2: Kalman Filter

Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



Example #2: Kalman Filter

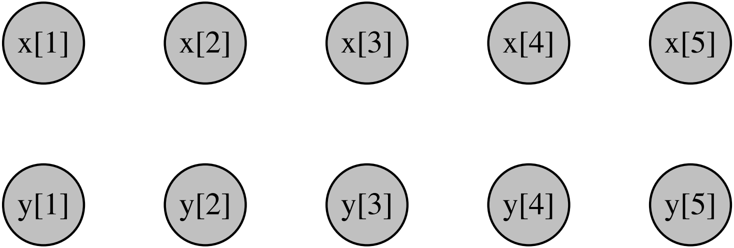
Code	Checkpoint
`x[1] ~ Gaussian(0.0, 1.0);`	
`y[1] ~ Gaussian(x[1], 1.0);`	
`for (t in 2..T) {`	
`  x[t] ~ Gaussian(a*x[t - 1], 1.0);`	
`  y[t] ~ Gaussian(x[t], 1.0);`	
`}`	
`stdout.print(x[1]);`	**value** x[1]



Example #2: Kalman Filter

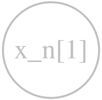
Code	**Checkpoint**

```
x[1] ~ Gaussian(0.0, 1.0);
y[1] ~ Gaussian(x[1], 1.0);
for (t in 2..T) {
  x[t] ~ Gaussian(a*x[t - 1], 1.0);
  y[t] ~ Gaussian(x[t], 1.0);
}
stdout.print(x[1]);
```

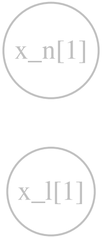


Example #3

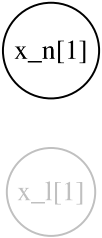
Example #3



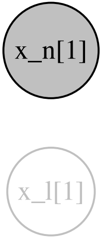
Example #3



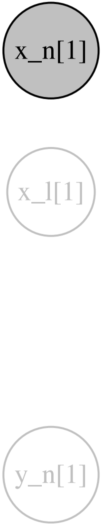
Example #3



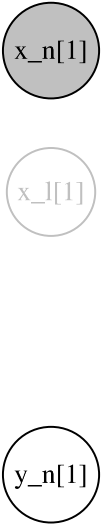
Example #3



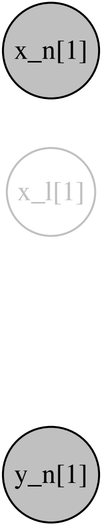
Example #3



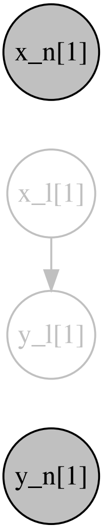
Example #3



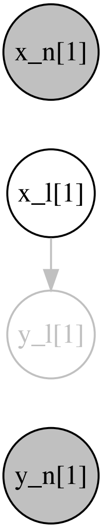
Example #3



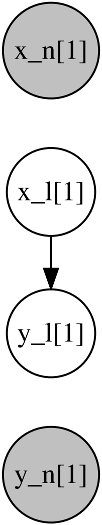
Example #3



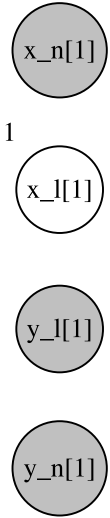
Example #3



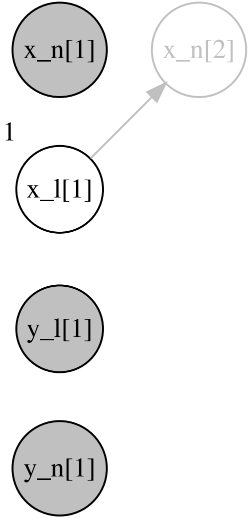
Example #3



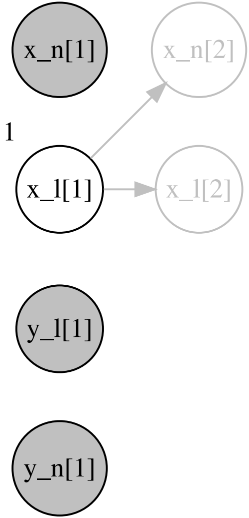
Example #3



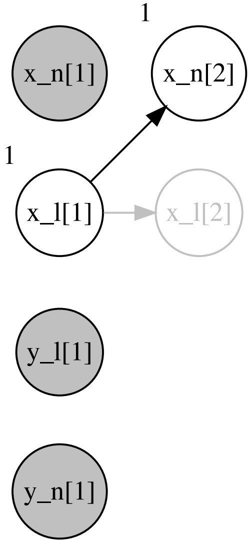
Example #3



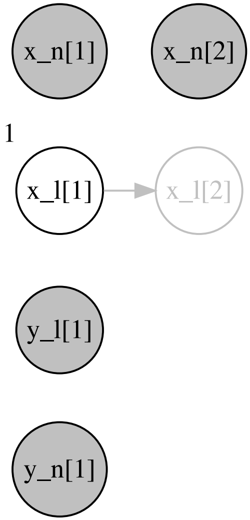
Example #3



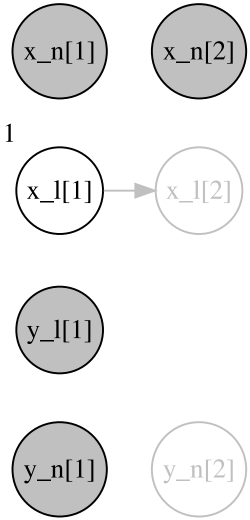
Example #3



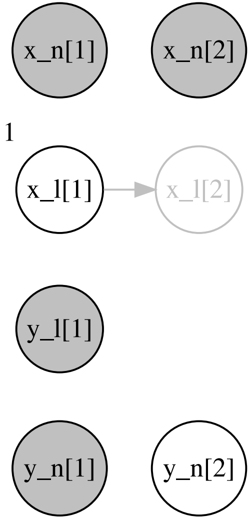
Example #3



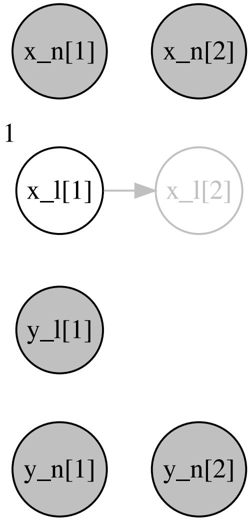
Example #3



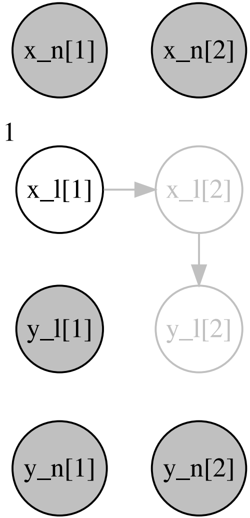
Example #3



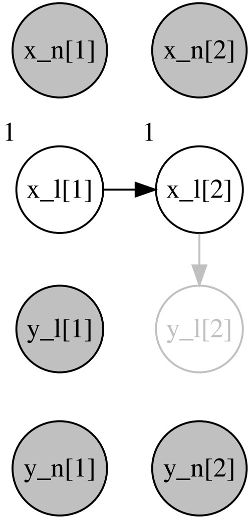
Example #3



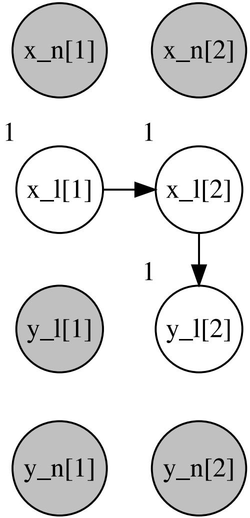
Example #3



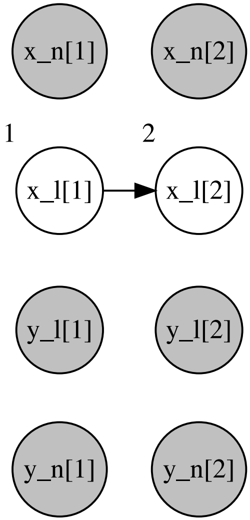
Example #3



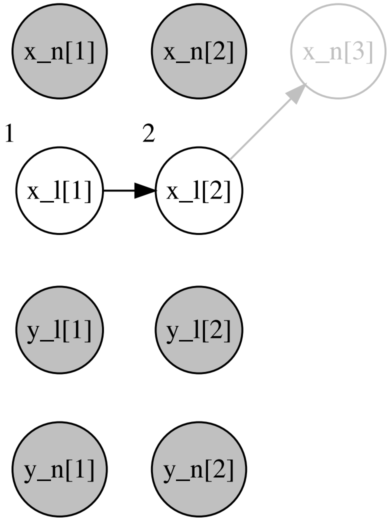
Example #3



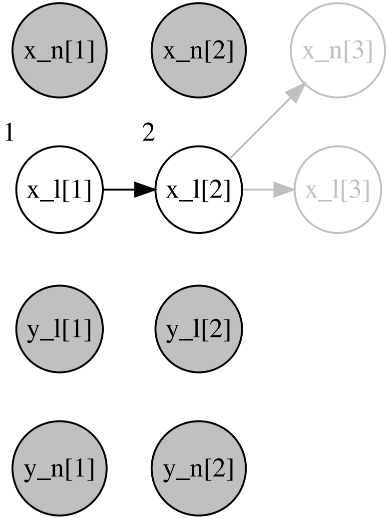
Example #3



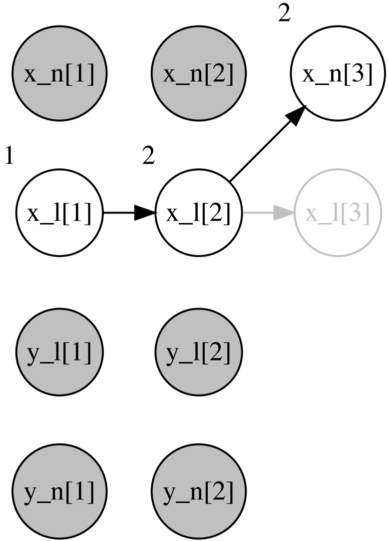
Example #3



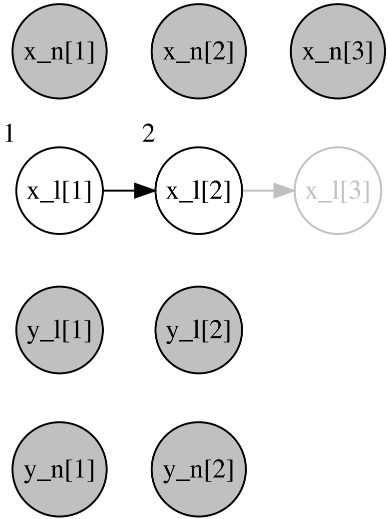
Example #3



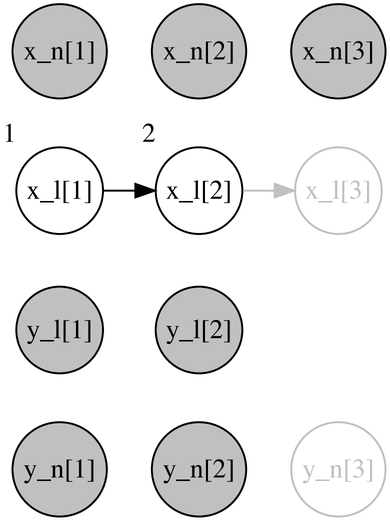
Example #3



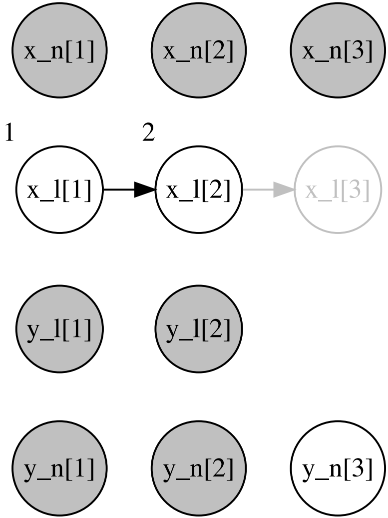
Example #3



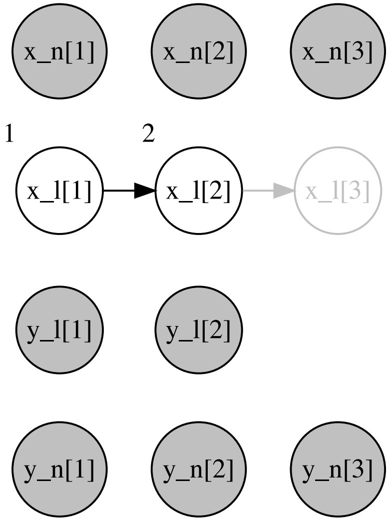
Example #3



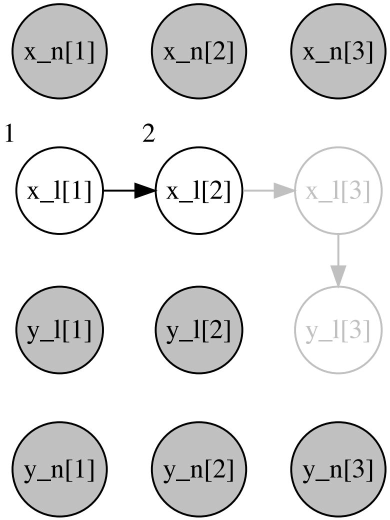
Example #3



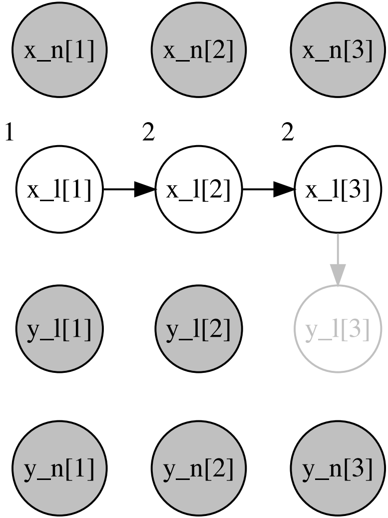
Example #3



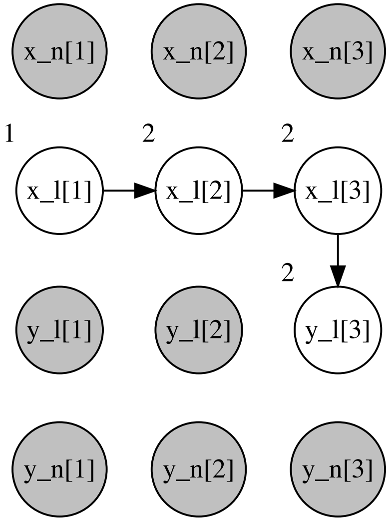
Example #3



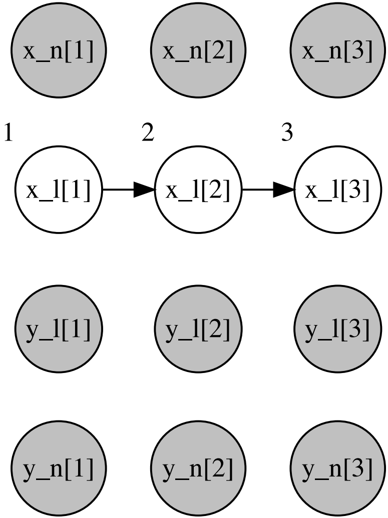
Example #3



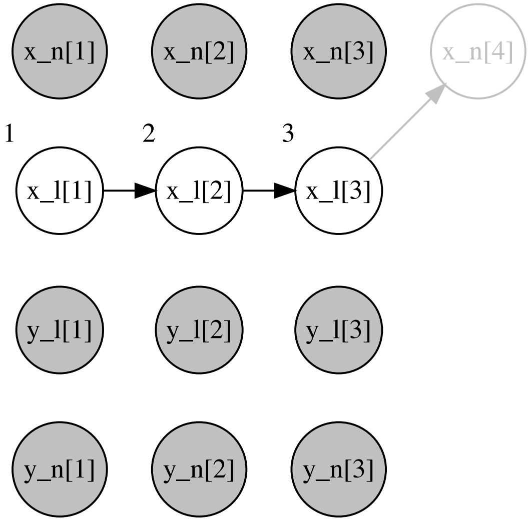
Example #3



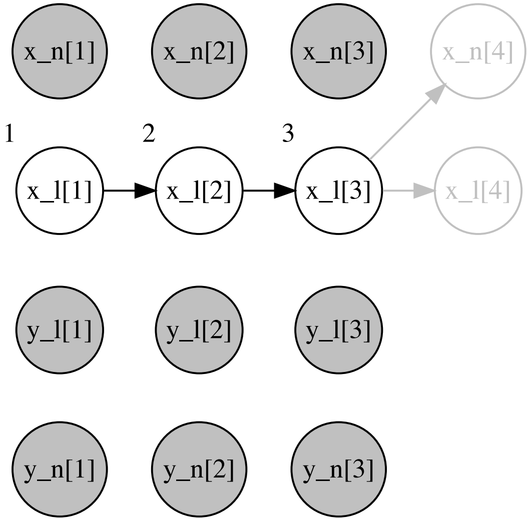
Example #3



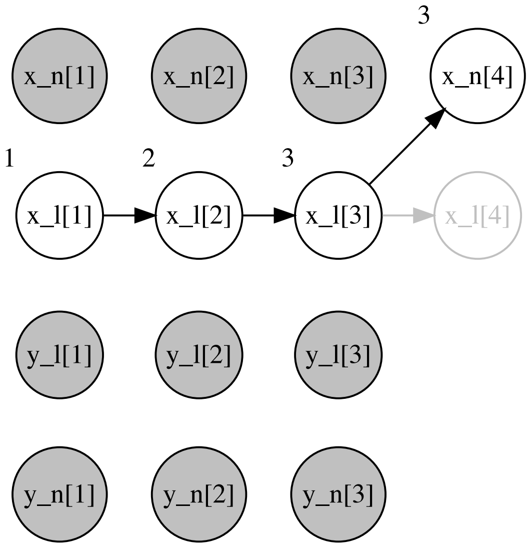
Example #3



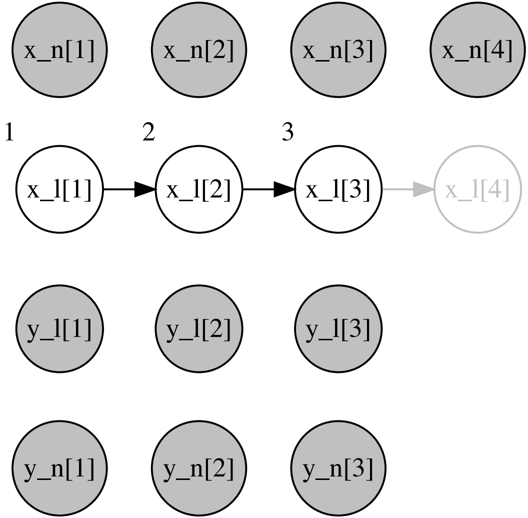
Example #3



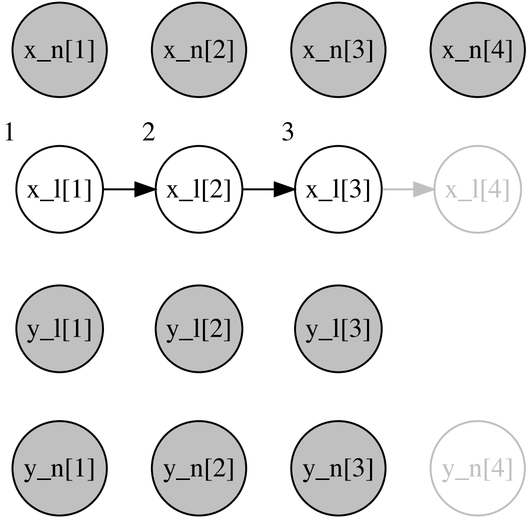
Example #3



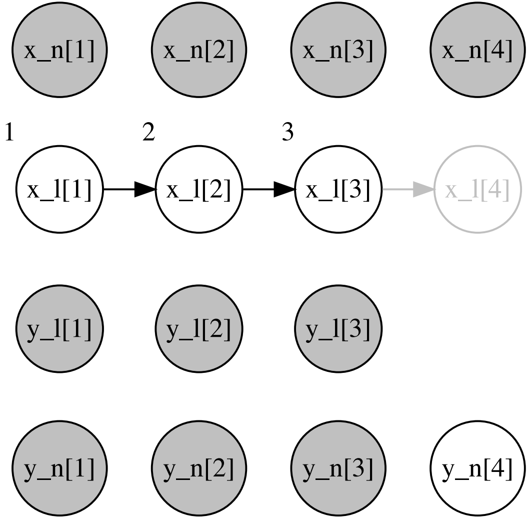
Example #3



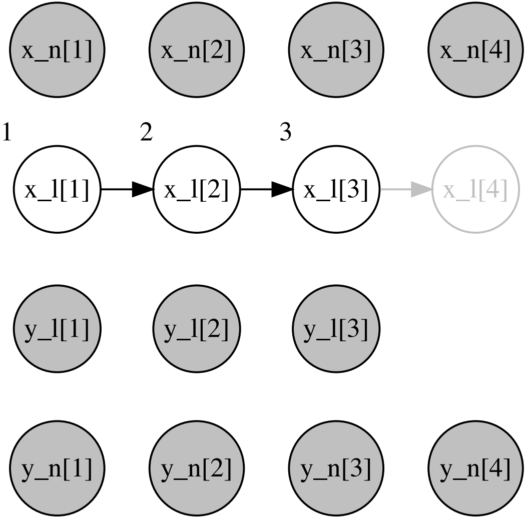
Example #3



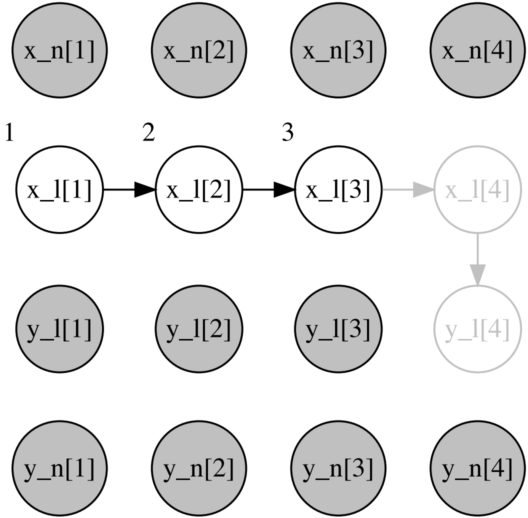
Example #3



Example #3

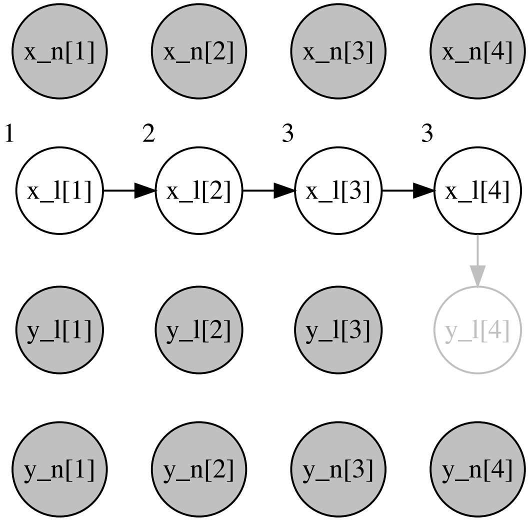


Example #3

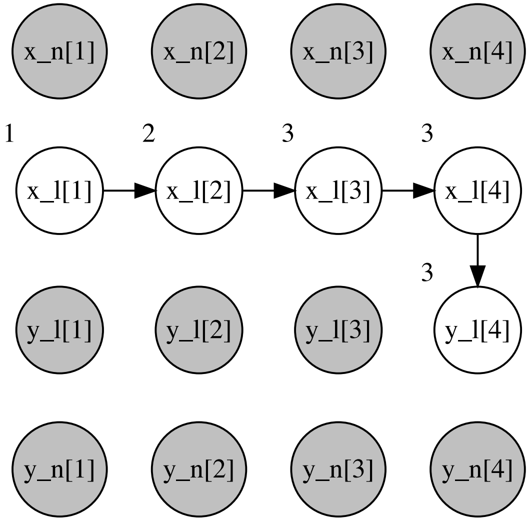


x_n[1]
x_n[2]
x_n[3]
x_n[4]
1
2
3
x_l[1]
x_l[2]
x_l[3]
x_l[4]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

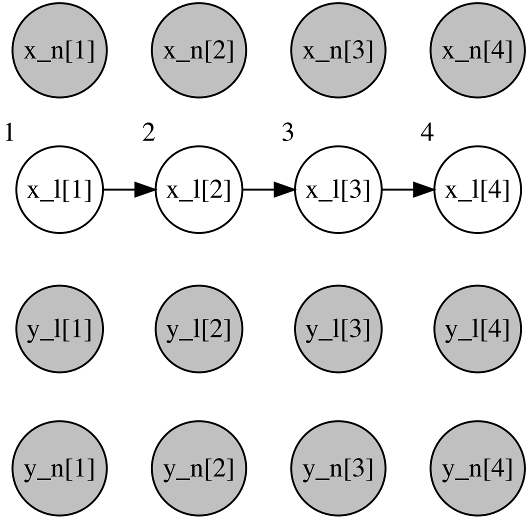
Example #3


x_n[1]
x_n[2]
x_n[3]
x_n[4]
1
2
3
3
x_l[1]
x_l[2]
x_l[3]
x_l[4]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

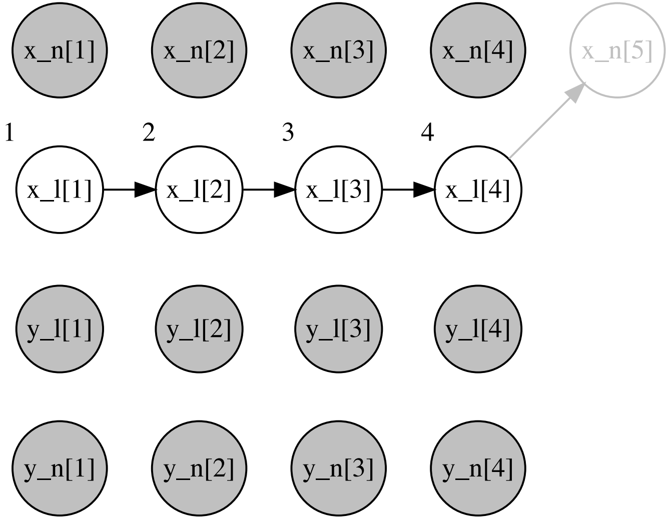
Example #3


x_n[1]
x_n[2]
x_n[3]
x_n[4]
1
2
3
3
x_l[1]
x_l[2]
x_l[3]
x_l[4]
3
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

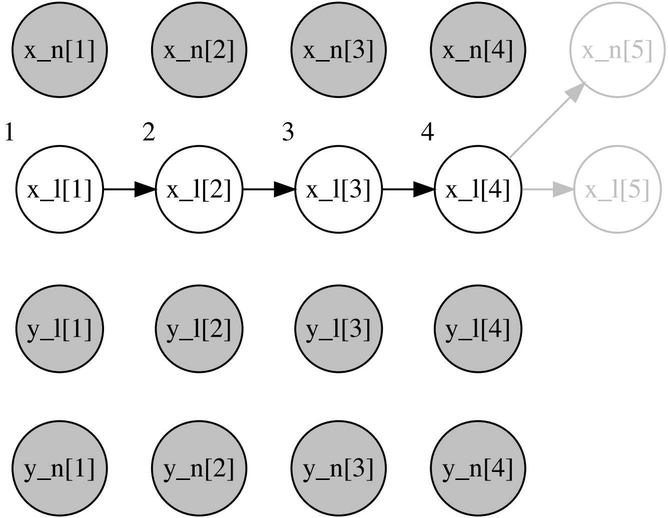
Example #3


x_n[1]
x_n[2]
x_n[3]
x_n[4]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

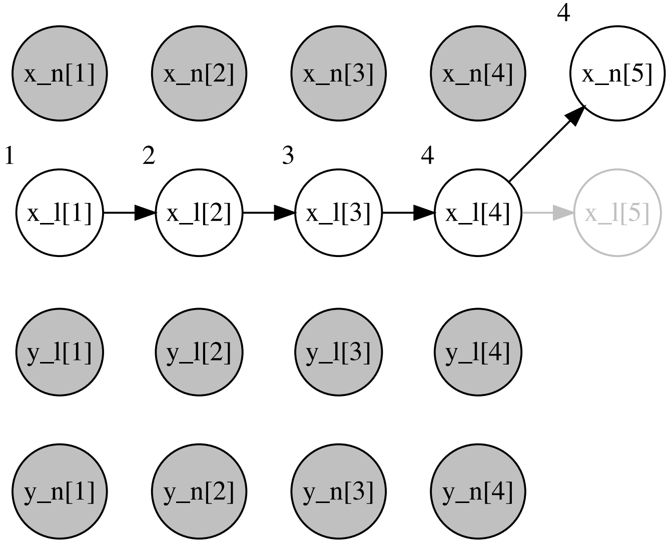
Example #3


x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

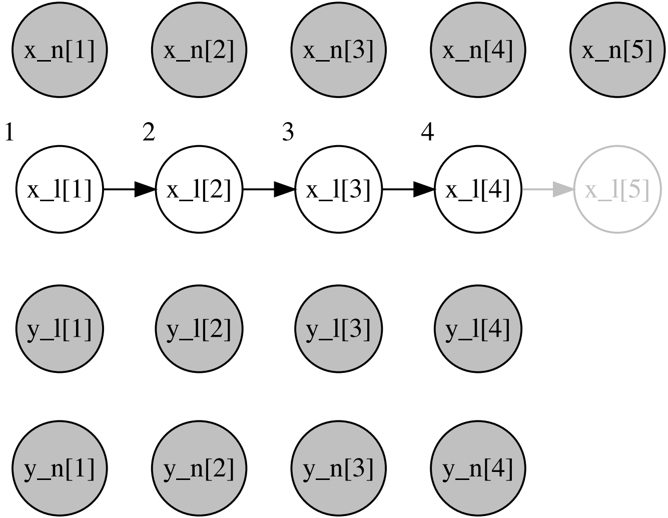
Example #3


x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

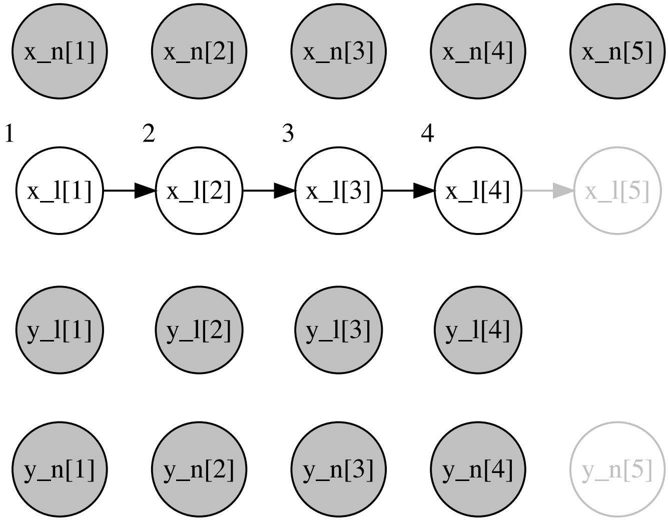
Example #3


4
x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

Example #3

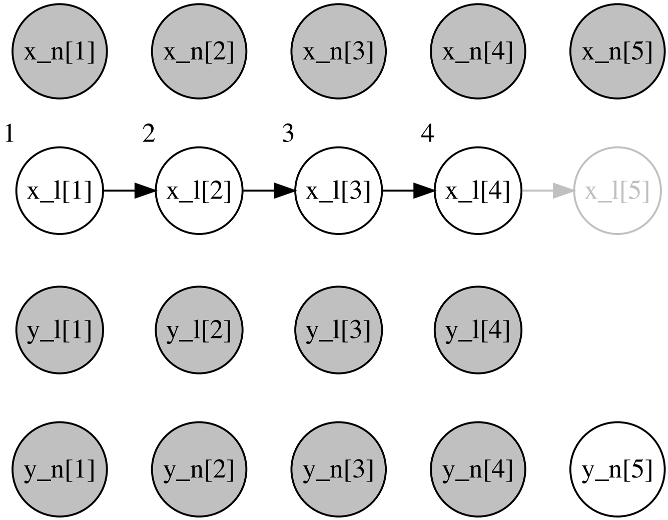

x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]

Example #3

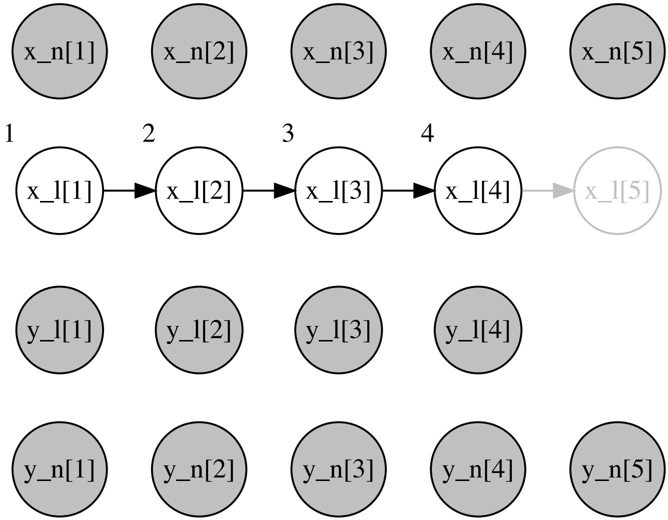


x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3

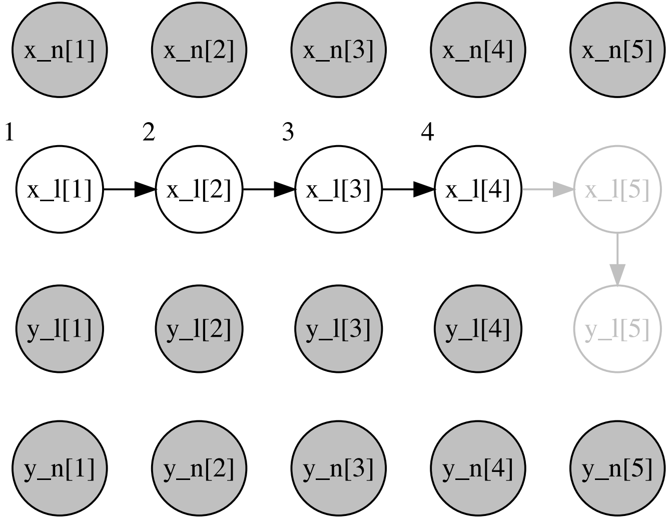

x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3

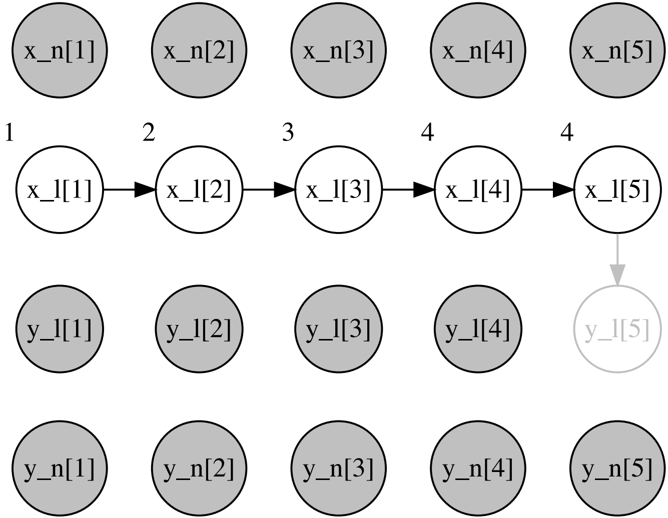


x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3

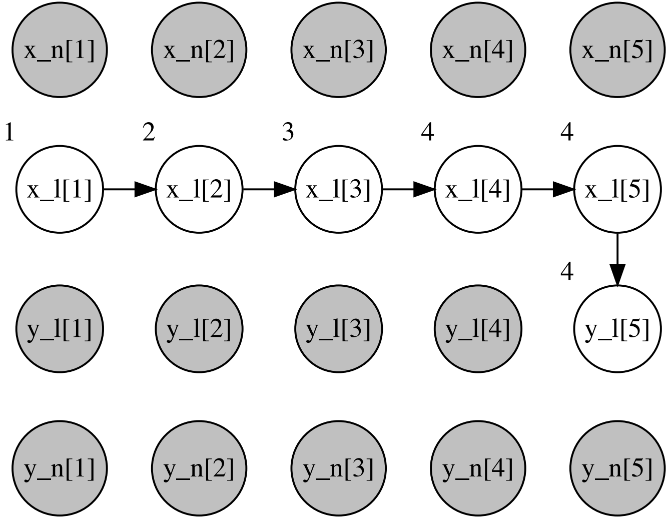

x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_l[5]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3



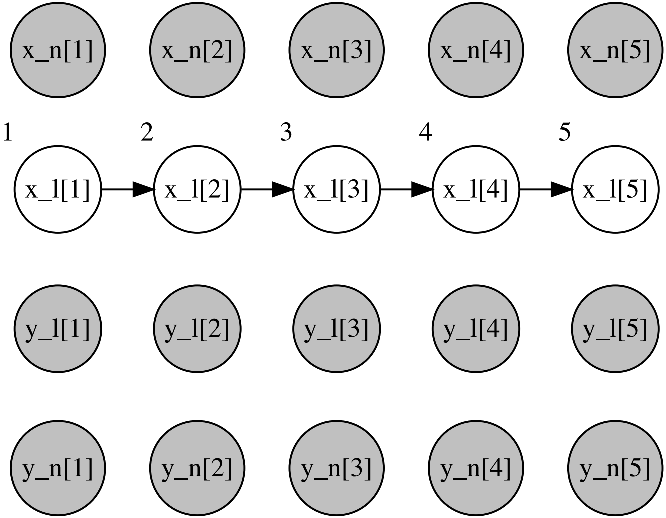
x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_l[5]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3

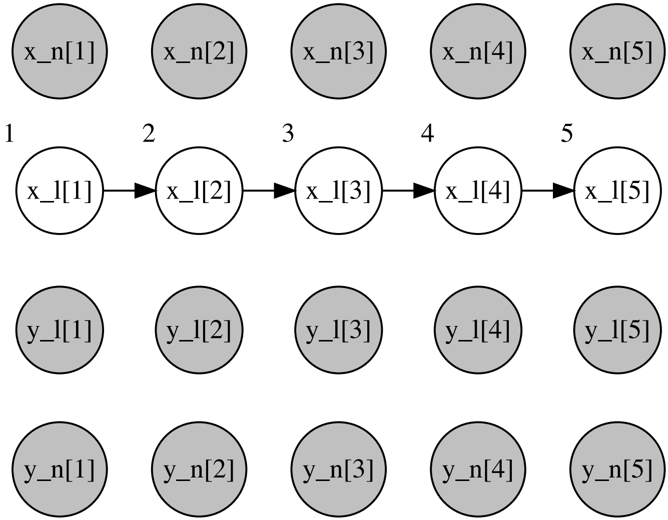


x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
4
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
4
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_l[5]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3


x_n[1]
x_n[2]
x_n[3]
x_n[4]
x_n[5]
1
2
3
4
5
x_l[1]
x_l[2]
x_l[3]
x_l[4]
x_l[5]
y_l[1]
y_l[2]
y_l[3]
y_l[4]
y_l[5]
y_n[1]
y_n[2]
y_n[3]
y_n[4]
y_n[5]

Example #3: Rao–Blackwellized Particle Filter



Programmatic models

1. Generative model of a joint distribution expressed in a universal programming language.

2. Countably infinite set of random variables $\{\mathsf{V}_\mathsf{k}\}_{\mathsf{k}=1}^{\infty}$.

3. As the model executes we encounter a finite subset of $\{\mathsf{V}_\mathsf{k}\}_{\mathsf{k}=1}^{\infty}$ in some order σ, producing a sequence $(\mathsf{V}_{\sigma[\mathsf{k}]})_{\mathsf{k}=1}^{|\sigma|}$, where $\sigma[\mathsf{k}]$ is given by a deterministic function Ne of the random variates so far: $\sigma[\mathsf{k}] = \mathsf{Ne}(\mathsf{v}_{\sigma[1]}, \ldots, \mathsf{v}_{\sigma[\mathsf{k}-1]})$.

4. When $\mathsf{V}_{\sigma[\mathsf{k}]}$ is encountered, it is associated with a distribution

$$\mathsf{V}_{\sigma[\mathsf{k}]} \sim \mathsf{p}_{\sigma[\mathsf{k}]}\left(\mathsf{dv}_{\sigma[\mathsf{k}]} \mid \mathsf{Pa}(\mathsf{v}_{\sigma[1]}, \ldots, \mathsf{v}_{\sigma[\mathsf{k}-1]})\right),$$

with Pa a deterministic function selecting a subset of its arguments.

Programmatic models

At some point execution terminates, having simulated

$$\mathsf{p}_\sigma(\mathsf{dv}_{\sigma[1]}, \ldots, \mathsf{dv}_{\sigma[|\sigma|]}) = \prod_{\mathsf{k}=1}^{|\sigma|} \mathsf{p}_{\sigma[\mathsf{k}]} \left(\mathsf{dv}_{\sigma[\mathsf{k}]} \mid \mathsf{Pa}(\mathsf{v}_{\sigma[1]}, \ldots, \mathsf{v}_{\sigma[\mathsf{k}-1]})\right).$$

Programmatic models

At some point execution terminates, having simulated

$$\mathsf{p}_\sigma(\mathsf{dv}_{\sigma[1]}, \ldots, \mathsf{dv}_{\sigma[|\sigma|]}) = \prod_{\mathsf{k}=1}^{|\sigma|} \mathsf{p}_{\sigma[\mathsf{k}]} \left(\mathsf{dv}_{\sigma[\mathsf{k}]} \mid \mathsf{Pa}(\mathsf{v}_{\sigma[1]}, \ldots, \mathsf{v}_{\sigma[\mathsf{k}-1]})\right).$$

We will be interested in executing the code several times. The nth execution will be associated with the distribution $\mathsf{p}_{\sigma_\mathsf{n}}$, given by

$$\mathsf{p}_{\sigma_\mathsf{n}}(\mathsf{dv}_{\sigma_\mathsf{n}[1]}, \ldots, \mathsf{dv}_{\sigma_\mathsf{n}[|\sigma_\mathsf{n}|]}) = \prod_{\mathsf{k}=1}^{|\sigma_\mathsf{n}|} \mathsf{p}_{\sigma_\mathsf{n}[\mathsf{k}]} \left(\mathsf{dv}_{\sigma_\mathsf{n}[\mathsf{k}]} \mid \mathsf{Pa}(\mathsf{v}_{\sigma_\mathsf{n}[1]}, \ldots, \mathsf{v}_{\sigma_\mathsf{n}[\mathsf{k}-1]})\right),$$

with $\sigma_\mathsf{n}[\mathsf{k}] = \mathsf{Ne}(\mathsf{v}_{\sigma_\mathsf{n}[1]}, \ldots, \mathsf{v}_{\sigma_\mathsf{n}[\mathsf{k}-1]})$. Subscript n is used to denote execution-dependent variables.

Programmatic models

For different executions n and m, it is possible for:

- the number of random variables encountered, $|\sigma_\mathsf{n}|$ and $|\sigma_\mathsf{m}|$, to differ,
- the sequences $(\mathsf{V}_{\sigma_\mathsf{n}[\mathsf{k}]})_{\mathsf{k}=1}^{|\sigma_\mathsf{n}|}$ and $(\mathsf{V}_{\sigma_\mathsf{m}[\mathsf{k}]})_{\mathsf{k}=1}^{|\sigma_\mathsf{m}|}$ to differ, and
- the two subsets $\{\mathsf{V}_{\sigma_\mathsf{n}[\mathsf{k}]}\}_{\mathsf{k}=2}^{|\sigma_\mathsf{n}|}$ and $\{\mathsf{V}_{\sigma_\mathsf{m}[\mathsf{k}]}\}_{\mathsf{k}=2}^{|\sigma_\mathsf{m}|}$ to be different (and even disjoint).

Programmatic models

For different executions n and m, it is possible for:

- the number of random variables encountered, $|\sigma_n|$ and $|\sigma_m|$, to differ,
- the sequences $(V_{\sigma_n[k]})_{k=1}^{|\sigma_n|}$ and $(V_{\sigma_m[k]})_{k=1}^{|\sigma_m|}$ to differ, and
- the two subsets $\{V_{\sigma_n[k]}\}_{k=2}^{|\sigma_n|}$ and $\{V_{\sigma_m[k]}\}_{k=2}^{|\sigma_m|}$ to be different (and even disjoint).

In general, p_{σ_n} and p_{σ_m} are not the same, but rather components of a mixture.

Programmatic models

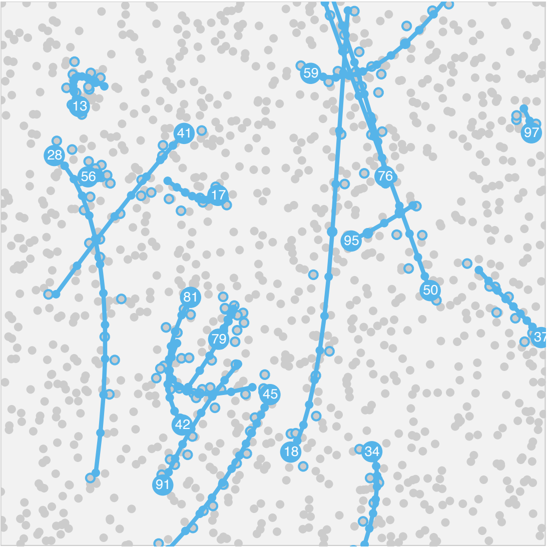
For different executions n and m, it is possible for:

- the number of random variables encountered, $|\sigma_\mathsf{n}|$ and $|\sigma_\mathsf{m}|$, to differ,
- the sequences $(\mathsf{V}_{\sigma_\mathsf{n}[\mathsf{k}]})_{\mathsf{k}=1}^{|\sigma_\mathsf{n}|}$ and $(\mathsf{V}_{\sigma_\mathsf{m}[\mathsf{k}]})_{\mathsf{k}=1}^{|\sigma_\mathsf{m}|}$ to differ, and
- the two subsets $\{\mathsf{V}_{\sigma_\mathsf{n}[\mathsf{k}]}\}_{\mathsf{k}=2}^{|\sigma_\mathsf{n}|}$ and $\{\mathsf{V}_{\sigma_\mathsf{m}[\mathsf{k}]}\}_{\mathsf{k}=2}^{|\sigma_\mathsf{m}|}$ to be different (and even disjoint).

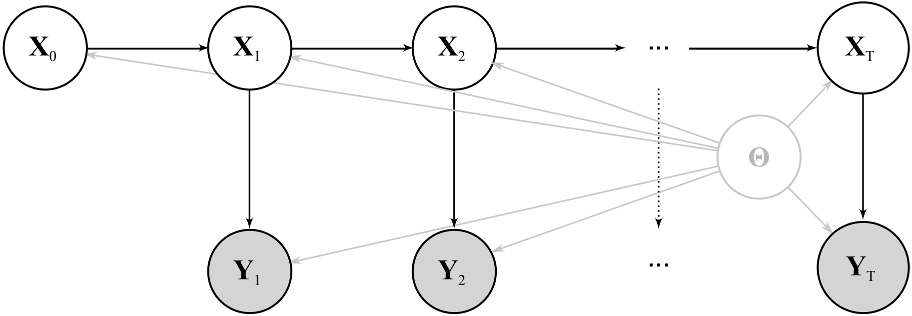
In general, $\mathsf{p}_{\sigma_\mathsf{n}}$ and $\mathsf{p}_{\sigma_\mathsf{m}}$ are not the same, but rather components of a mixture.

Models that fall in the programmatic class but not the graphical class occur in e.g. nonparametrics, phylogenetics, computational linguistics, multiple object tracking.

Multiple object tracking

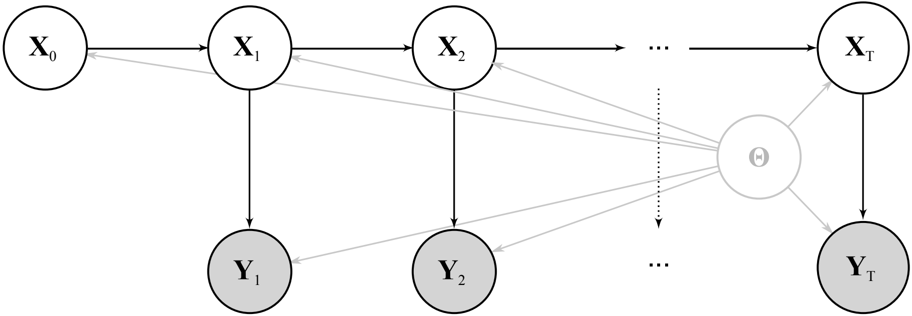


Multiple object tracking



$$\underbrace{p(dy_{1:T}, dx_{0:T}, d\theta)}_{\text{joint}} = \underbrace{p(d\theta)}_{\text{parameter}} \underbrace{p(dx_0 \mid \theta)}_{\text{initial}} \prod_{t=1}^{T} \underbrace{p(dx_t \mid x_{t-1}, \theta)}_{\text{transition}} \underbrace{p(dy_t \mid x_t, \theta)}_{\text{observation}}$$

Multiple object tracking



$$\underbrace{p(dy_{1:T}, dx_{0:T}, d\theta)}_{\text{joint}} = \underbrace{p(d\theta)}_{\text{parameter}} \underbrace{p(dx_0 \mid \theta)}_{\text{initial}} \prod_{t=1}^{T} \underbrace{p(dx_t \mid x_{t-1}, \theta)}_{\text{transition}} \underbrace{p(dy_t \mid x_t, \theta)}_{\text{observation}}$$

But random state size, random observation size, random association between them.

Single object model

The single object model is a linear-Gaussian state-space model. For the ith object:

$$
\begin{aligned}
X_0^i &\sim \mathcal{N}(\mu_0^i, M) \\
X_t^i &\sim \mathcal{N}(Ax_{t-1}^i, Q) \\
D_t^i &\sim \text{Bernoulli}(\rho) && \text{(is the object detected?)} \\
Y_t^i &\sim \mathcal{N}(Bx_t^i, R) && \text{(observation if detected.)}
\end{aligned}
$$

Single object model

```
class Track < StateSpaceModel<Global,Random<Real[_]>,Random<Real[_]>> {
  t:Integer;  // starting time of the track

  fiber initial(x:Random<Real[_]>, θ:Global) -> Real {
    auto μ <- vector(0.0, 3*length(θ.l));
    μ[1..2] <~ Uniform(θ.l, θ.u);
    x ~ Gaussian(μ, θ.M);
  }

  fiber transition(x':Random<Real[_]>, x:Random<Real[_]>, θ:Global) -> Real {
    x' ~ Gaussian(θ.A*x, θ.Q);
  }

  fiber observation(y:Random<Real[_]>, x:Random<Real[_]>, θ:Global) -> Real {
    d:Boolean;
    d <~ Bernoulli(θ.d);  // is the track detected?
    if d {
      y ~ Gaussian(θ.B*x, θ.R);
    }
  }
}
```

Multiple object model

At time t:

- the number of new objects is

$$B_t \sim \text{Poisson}(\lambda),$$

- the lifetime of each new object i is

$$S^i \sim \text{Poisson}(\tau),$$

  so that the probability of an object disappearing if it has been present for s^i time steps is $\Pr[S^i = s^i]/\Pr[S^i \geq s^i]$,
- the number of spurious observations (clutter) C_t is

$$C_t - 1 \sim \text{Poisson}(\mu),$$

  with these uniformly distributed on the domain $[l, u]$.

Multiple object model

```
class Multi < StateSpaceModel<Global,List<Track>,List<Random<Real[_]>>> {
  t:Integer <- 0;  // current time

  fiber transition(x':List<Track>, x:List<Track>, θ:Global) -> Real {
    t <- t + 1;

    /* move current objects */
    auto track <- x.walk();
    while track? {
      ρ:Real <- pmf_poisson(t - track!.t - 1, θ.τ);
      R:Real <- 1.0 - cdf_poisson(t - track!.t - 1, θ.τ) + ρ;
      s:Boolean;
      s <~ Bernoulli(1.0 - ρ/R);  // does the object survive?
      if s {
        track!.step();
        x'.pushBack(track!);
      }
    }
    ...
```

Multiple object model

```
    ...

    /* birth new objects */
    N:Integer;
    N <~ Poisson(θ.λ);
    for n:Integer in 1..N {
      track:Track;
      track.t <- t;
      track.θ <- θ;
      track.start();
      x'.pushBack(track);
    }
  }
  ...
```

Multiple object model

```
  ...

  fiber observation(y:List<Random<Real[_]>>, x:List<Track>, θ:Global) -> Real {
    if !y.empty() {  // observations given, use data association
      association(y, x, θ);
    } else {
      N:Integer;
      N <~ Poisson(θ.μ);
      for n:Integer in 1..(N + 1) {
        clutter:Random<Real[_]>;
        clutter <~ Uniform(θ.l, θ.u);
        y.pushBack(clutter);
      }
    }
  }
}
```

Inference

- There is **structure** to leverage: the whole model consists of multiple state-space models for single objects within a larger state-space model for managing multiple objects.

- There is **form** to leverage: the inner state-space models are linear-Gaussian.

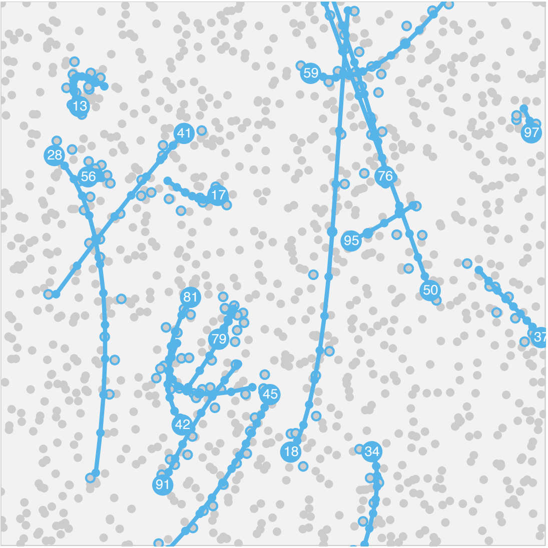
Inference

- There is **structure** to leverage: the whole model consists of multiple state-space models for single objects within a larger state-space model for managing multiple objects.

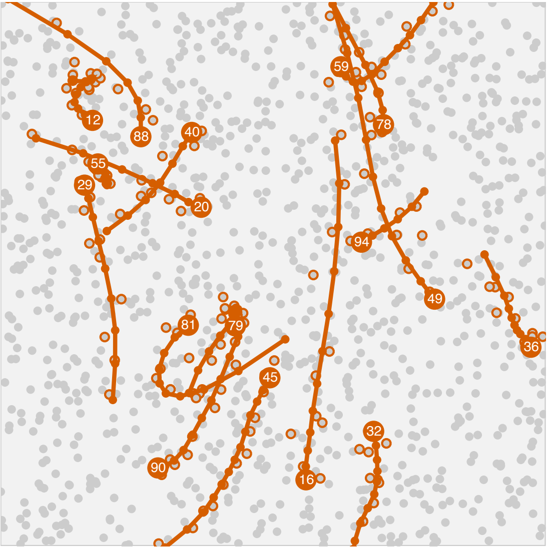
- There is **form** to leverage: the inner state-space models are linear-Gaussian.

If we run a particle filter on the outer state-space model, delayed sampling gives us, within each particle, a separate Kalman filter on the inner state-space models.

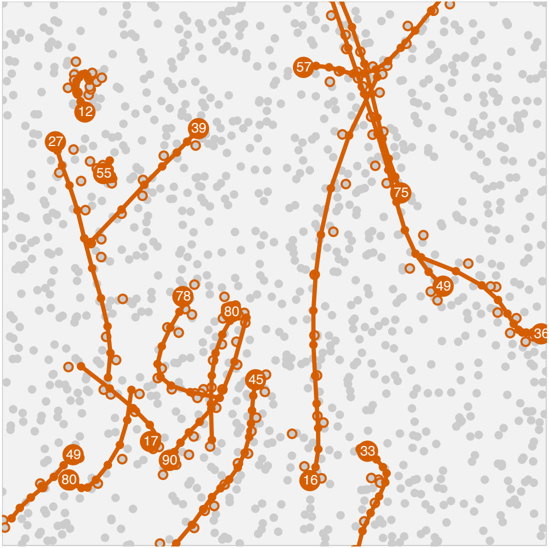
Multiple object tracking



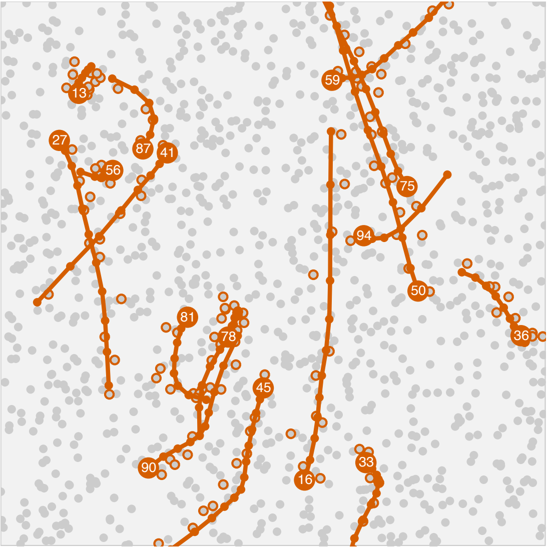
Multiple object tracking



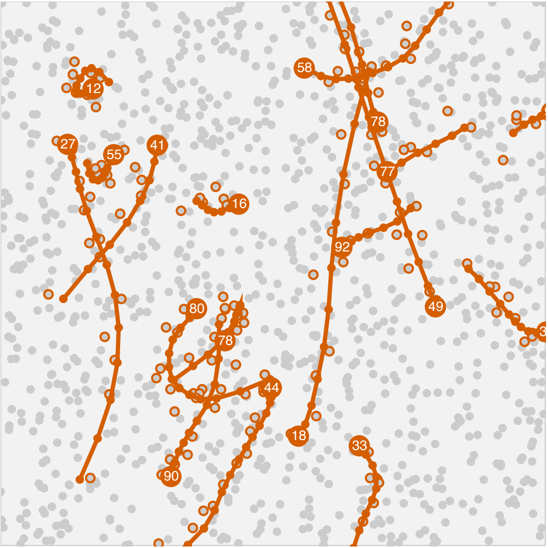
Multiple object tracking



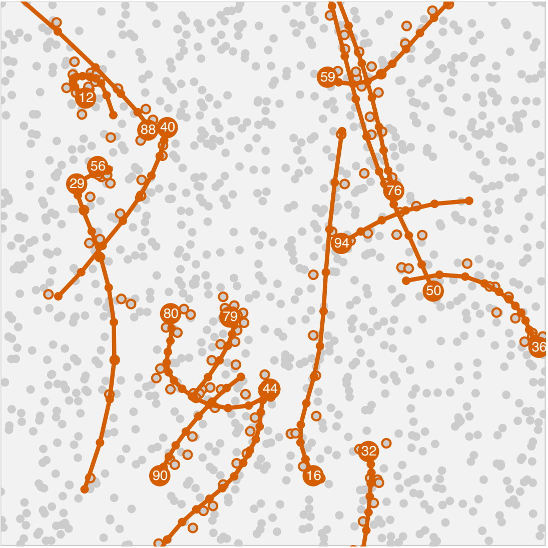
Multiple object tracking



Multiple object tracking



Multiple object tracking



Data association

```
fiber association(y:List<Random<Real[_]>>, x:List<Track>, θ:Global) -> Real {
  K:Integer <- 0;  // number of detections
  auto track <- x.walk();
  while track? {
    if track!.y.back().hasDistribution() {
      /* object is detected, compute proposal */
      K <- K + 1;
      q:Real[y.size()];
      n:Integer <- 1;
      auto detection <- y.walk();
      while detection? {
        q[n] <- track!.y.back().pdf(detection!);
        n <- n + 1;
      }
      Q:Real <- sum(q);
      ...
```

Multiple object model

```
    ...

    /* propose an association */
    if Q > 0.0 {
      q <- q/Q;
      n <~ Categorical(q);  // choose an observation
      yield track!.y.back().realize(y.get(n));  // likelihood
      yield -log(q[n]);  // proposal correction
      y.erase(n);  // remove the observation for future associations
    } else {
      yield -inf;  // detected, but all likelihoods (numerically) zero
    }
  }

  /* factor in prior probability of hypothesis */
  yield -lrising(y.size() + 1, K);  // prior correction
}
...
```

Multiple object model

```
  ...

  /* clutter */
  y.size() - 1 ~> Poisson(θ.μ);
  auto clutter <- y.walk();
  while clutter? {
    clutter! ~> Uniform(θ.l, θ.u);
  }
}
```

Summary

Getting started guide and tutorial available on the website: birch-lang.org.

Papers

- L. M. Murray and T. B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, to appear, 2018. URL arxiv.org/abs/1810.01539
- L. M. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. B. Schön. Delayed sampling and automatic Rao–Blackwellization of probabilistic programs. In *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS)*, Lanzarote, Spain, 2018. URL arxiv.org/abs/1708.07787