CS6202: Probabilistic Programming for Bayesian Machine Learning

Generative Modelling

Luke Ong

What is a model?

1. A probability density or mass function.
2. A simulator or a sampler.

What is a model?

1. A probability density or mass function.
2. A simulator or a sampler.
   - Generative view.
   - Well-supported by PPLs like Anglican.
   - Express how a hypothesis arises and also how it generates observed data.

Solving a problem via generative modelling

1. Create a simulator (i.e. a generative model).
   - How do candidate solutions arise?
   - How do solutions generate observations?
2. Specify success criteria with conditioning.
3. Solve the problem by posterior inference.

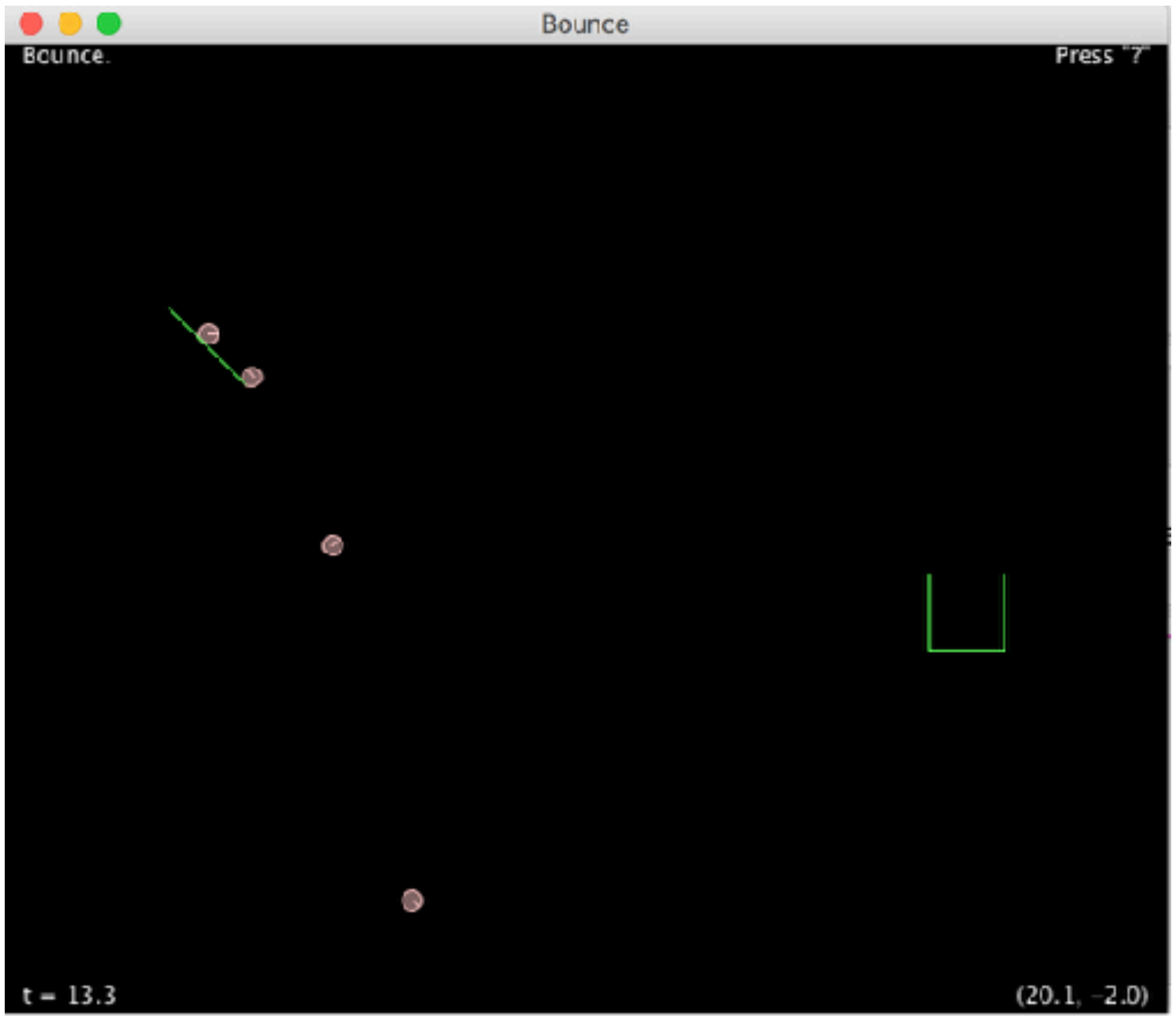
Learning outcome

1. Can use generative modelling for solving a problem.
2. Can reproduce the solutions of 2D physics and program induction in Anglican.
3. Can write interesting probabilistic programs using the expressive power of Anglican.
4. Can explain Poisson, and Gaussian distributions.

2D physics

Based on Wood & Paige's practical at MLSS'15

2D physics

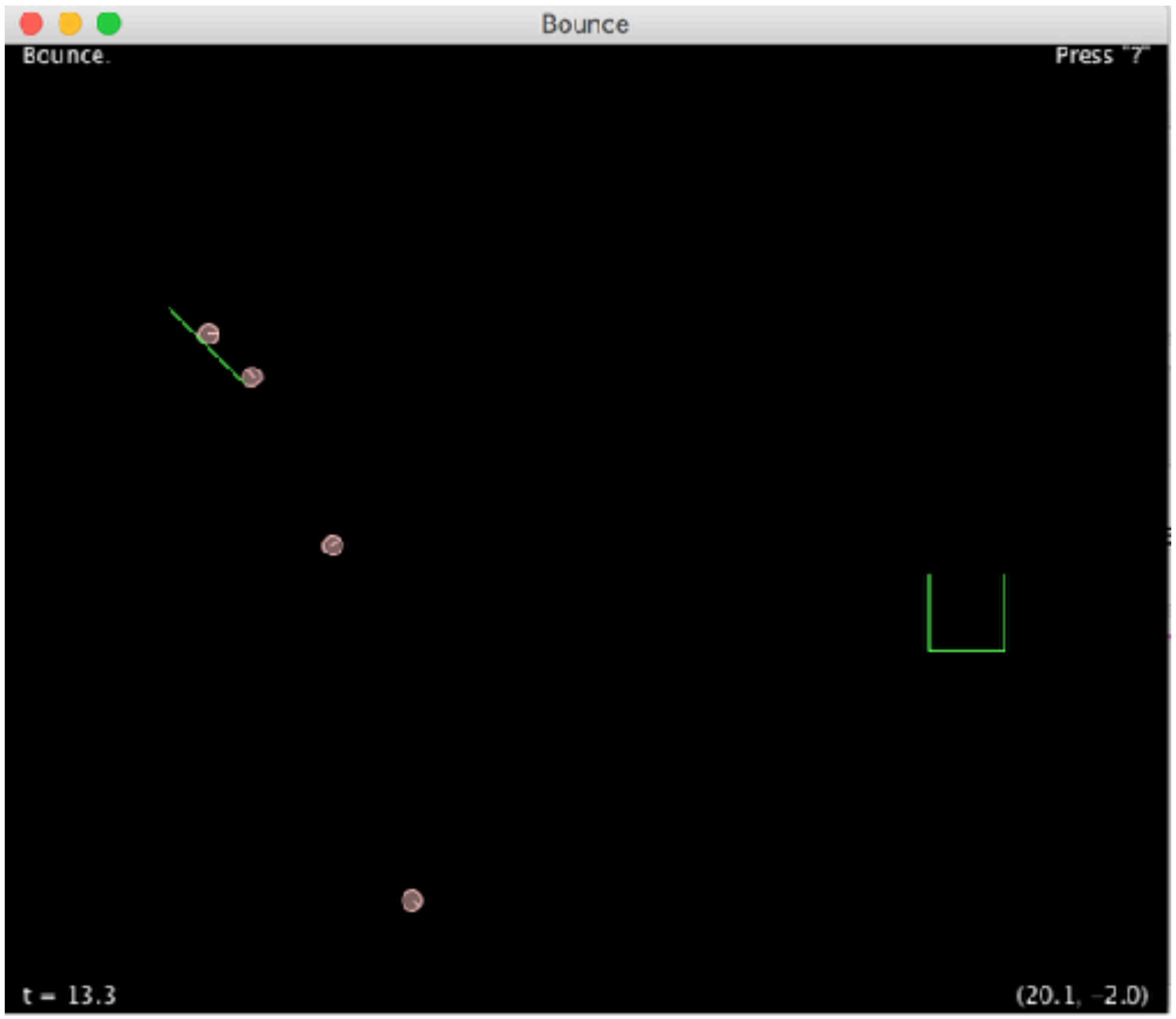


20 balls are falling from top-left.

[Q] How to move them to the bin?

Borrowed from Wood & Paige's practical at MLSS'15

2D physics



20 balls are falling from top-left.

[Q] How to move them to the bin?

[A] Put bumpers.

Borrowed from Wood & Paige's practical at MLSS'15

How to solve this in Anglican?

(by means of generative modelling)

How to solve this in Anglican?

(by means of generative modelling)

How to solve this in Anglican?

1. Create a simulator.

2. Specify success criteria with conditioning.

3. Solve the problem by posterior inference.

(by means of generative modelling)

How to solve this in Anglican?

1. Create a simulator.

2. Specify success criteria with conditioning.

3. Solve the problem by posterior inference.

(by means of generative modelling)

How to solve this in Anglican?

1. Create a simulator.
    - Randomly place bumpers. Drop balls.
2. Specify success criteria with conditioning.
3. Solve the problem by posterior inference.

(by means of generative modelling)

How to solve this in Anglican?

1. Create a simulator.
   - Randomly place bumpers. Drop balls.
2. Specify success criteria with conditioning.
3. Solve the problem by posterior inference.

(by means of generative modelling)

How to solve this in Anglican?

1. Create a simulator.
    - Randomly place bumpers. Drop balls.
2. Specify success criteria with conditioning.
    - High likelihood if many balls reach the bin.
3. Solve the problem by posterior inference.

(by means of generative modelling)

How to solve this in Anglican?

1. Create a simulator.
   - Randomly place bumpers. Drop balls.
2. Specify success criteria with conditioning.
   - High likelihood if many balls reach the bin.
3. Solve the problem by posterior inference.

Clojure functions for 2D physics in bounce.clj

1. Create a world.

```
(def bumpers (list (list -3 6) (list 2 5)))
(def start-w (create-world bumpers))
```

2. Simulate 20 balls in the world.

```
(def end-w (simulate-world start-w))
```

3. Count the number of balls in the bin.

```
(def num-balls (balls-in-box end-w))
```

These functions are defined in bounce.clj available at the course webpage.

Clojure functions in Anglican programs

```
(with-primitive-procedures
    [create-world
     simulate-world
     balls-in-box]

  (defquery physics [] . . . ))
```

Clojure functions in Anglican programs

```
(with-primitive-procedures
    [create-world
     simulate-world
     balls-in-box]

  (defquery physics [] . . . ))
```

Anglican keyword.
Imports primitive Clojure fns.

Clojure functions in Anglican programs

```
(with-primitive-procedures
    [create-world
     simulate-world
     balls-in-box]

  (defquery physics [] . . . ))
```

Imported functions.

Anglican keyword.
Imports primitive Clojure fns.

1) Simulator. 2) Condition. 3) Inference.

1) Simulator. ~~2) Condition.~~ 3) Inference.

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(defquery physics0 []

          •    •    •    •    •

                          )
```

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics0 []

            ● ● ● ● ●

                             ))
```

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics0 []
    (let
      [n-bumpers 8
       f (fn [] (list
            (sample (uniform-continuous -5 14))
            (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)


       . . . . .                  ]


     ● ● ● ● ●        )))
```

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics0 []
    (let
      [n-bumpers 8
       f (fn [] (list
             (sample (uniform-continuous -5 14))
             (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]


      • • • • •       )))
```

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics0 []
    (let
      [n-bumpers 8
       f (fn [] (list
           (sample (uniform-continuous -5 14))
           (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]


      (list num-balls bs))))
```

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(with primitive procedures
```

```
(def lazy-samples0
  (doquery :importance physics0 []))
(def samples0
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples0)))))
(def best-sample0
  (reduce (fn [acc x] (if (> (first x) (first acc)) x acc))
          samples0))
best-sample0
```

```
#'bounce-worksheet/lazy-samples0

#'bounce-worksheet/samples0

#'bounce-worksheet/best-sample0

(2 ((5.039659490241706 2.4748229831103163) (1.6105279484809571
6.095328821668973) (-1.7327828746932634 2.1429922008512325)
(11.46787625011067 3.6077249028398284) (-1.1506281530451017
2.1718715228712937) (-1.752497599843685 5.686404684681266)
(0.3924210062883362 4.924024324154887) (13.669929061656298
3.9039578861003066)))
```

1) Simulator. ~~2) Condition.~~ 3) Inference.

```
(def lazy-samples0
  (doquery :importance physics0 []))
(def samples0
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples0)))))
(def best-sample0
  (reduce (fn [acc x] (if (> (first x) (first acc)) x acc))
          samples0))
best-sample0
```

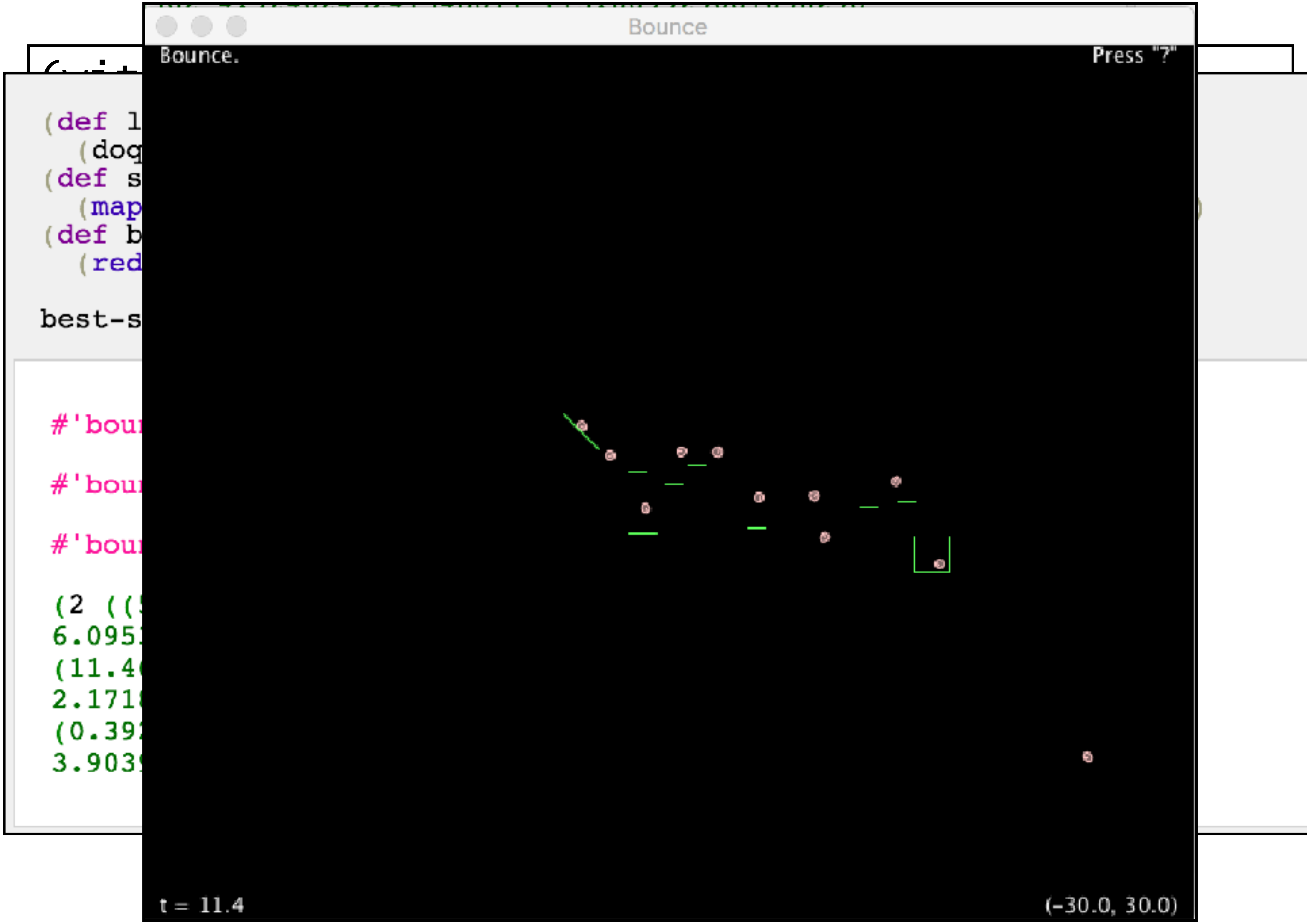
```
#'bounce-worksheet/lazy-samples0

#'bounce-worksheet/samples0
```

Just two balls in the bin.

```
(2 ((5.039659490241706 2.4748229831103163) (1.6105279484809571
6.095328821668973) (-1.7327828746932634 2.142992200851232 5)
(11.467876250110 67 3.6077249028398284) (-1.150628153045101 7
2.171871522871293 7) (-1.752497599843685 5.686404684681266)
(0.3924210062883362 4.924024324154887) (13.669929061656298
3.9039578861003066)))
```

1) Simulator. ~~2) Condition.~~ 3) Inference.



1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics1 []
    (let
      [n-bumpers 8
       f (fn [] (list
             (sample (uniform-continuous -5 14))
             (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]


      (list num-balls bs))))
```

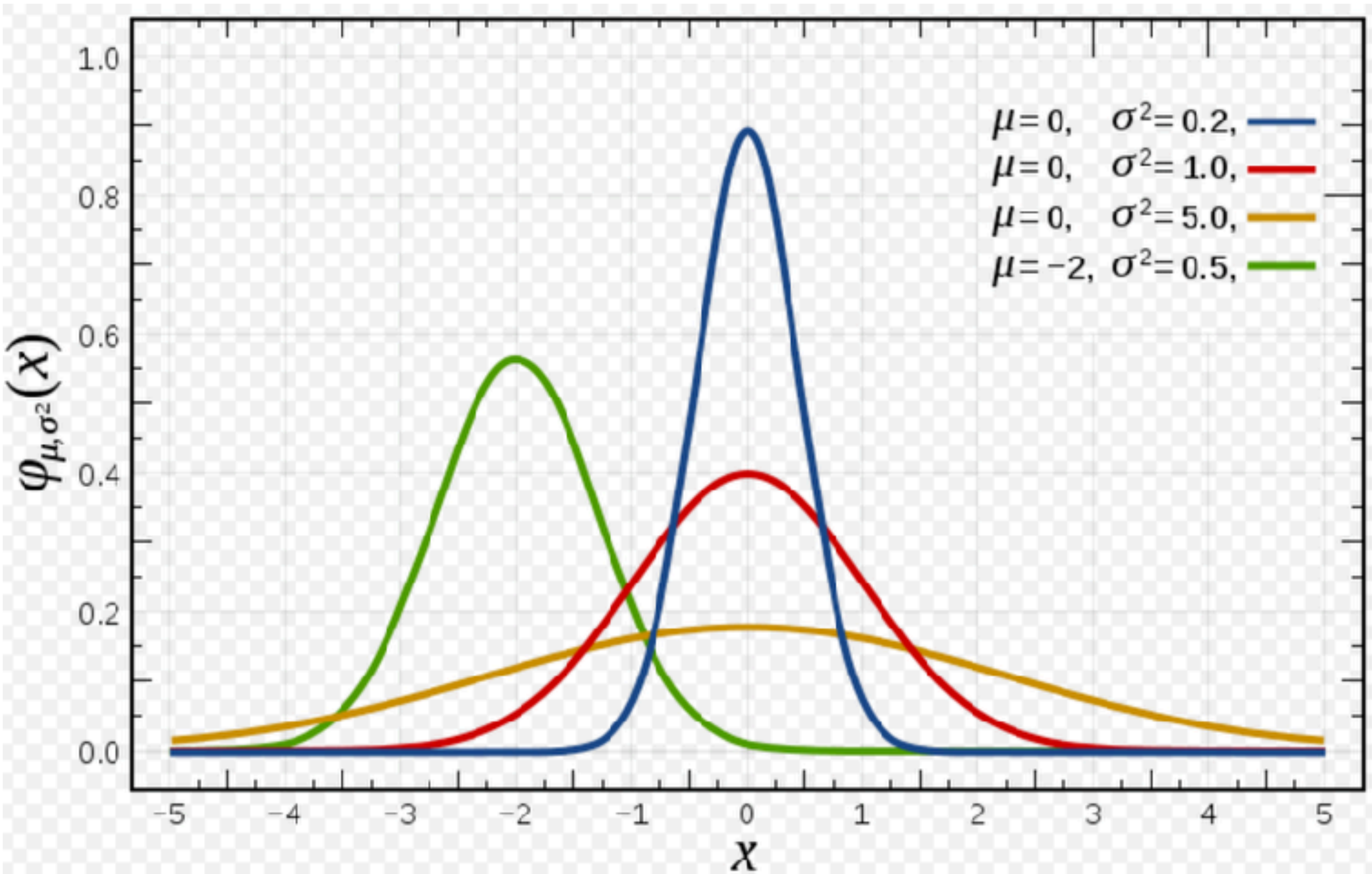
[Q] Want more balls in the bin. Express this goal using observe.

Normal distribution

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Probability distribution on a real x.
- Specified by density f. Also by μ and σ.

Normal distribution

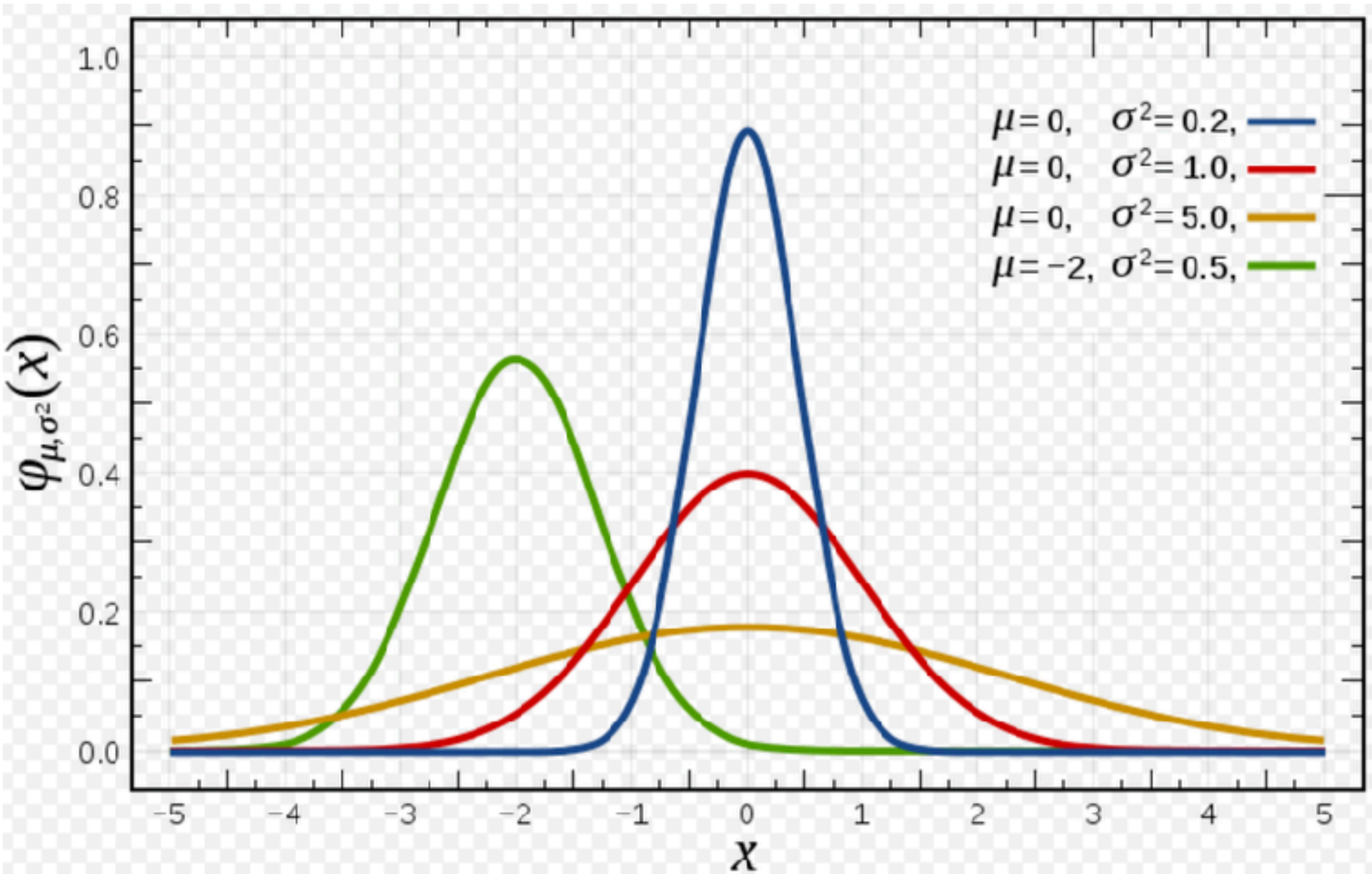


Normal distribution

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Probability distribution on a real x.
- Specified by density f. Also by μ and σ.
- [Intuition 1] x is in [μ-2σ, μ+2σ] usually.
- [Intuition 2] The chance of x being further from μ decays exponentially.

Normal distribution



Normal distribution

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$p(x \mid y) = \frac{p(y \mid x) \times p(x)}{p(y)}$$

- Bayes' rule holds regardless of whether p is a probability mass or a probability density.

1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics1 []
    (let
      [n-bumpers 8
       f (fn [] (list
           (sample (uniform-continuous -5 14))
           (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]



      (list num-balls bs))))
```

[Q] Want more balls in the bin. Express this goal using observe.

1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics1 []
    (let
      [n-bumpers 8
       f (fn [] (list
             (sample (uniform-continuous -5 14))
             (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]

      (observe (normal num-balls 1) 20)
      (list num-balls bs))))
```

[Q] Want more balls in the bin. Express this goal using observe.

1) Simulator. 2) Condition. 3) Inference.

```
(def lazy-samples1
  (doquery :lmh physics1 []))
(def samples1
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples1)))))
(def best-sample1
  (reduce (fn [acc x] (if (> (first x) (first acc)) x acc))
          samples1))
best-sample1
```

```
#'bounce-worksheet/lazy-samples1

#'bounce-worksheet/samples1

#'bounce-worksheet/best-sample1

(18 ((0.42846830943289227 4.955430158101355) (0.11399011311342733
7.047454084589595) (-1.6760284246021682 3.482088149196809)
(2.5215232534350247 2.5347454541889936) (12.119916165254827
7.446364186029621) (9.842132376151833 0.6555153748336084)
(0.383232923290564 0.8118697300596889) (5.731228040496732
5.245105763879869)))
```

[Q] Want more balls in the bin. Express this goal using observe.

1) Simulator. 2) Condition. 3) Inference.

MCMC algorithm (next lecture)

```
(def lazy-samples1
  (doquery :lmh physics1 []))
(def samples1
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples1)))))
(def best-sample1
  (reduce (fn [acc x] (if (> (first x) (first acc)) x acc))
          samples1))
best-sample1
```

```
#'bounce-worksheet/lazy-samples1

#'bounce-worksheet/samples1

#'bounce-worksheet/best-sample1

(18 ((0.42846830943289227 4.955430158101355) (0.11399011311342733
7.047454084589595) (-1.6760284246021682 3.482088149196809)
(2.521523253435024 7 2.534745454188993 6) (12.119916165254827
7.446364186029621) (9.842132376151833 0.6555153748336084)
(0.38323292329056 4 0.8118697300596889) (5.731228040496732
5.245105763879869)))
```

[Q] Want more balls in the bin. Express this goal using observe.

1) Simulator. 2) Condition. 3) Inference.

```
(def lazy-samples1
  (doquery :lmh physics1 []))
(def samples1
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples1)))))
(def best-sample1
  (reduce (fn [acc x] (if (> (first x) (first acc)) x acc))
          samples1))
best-sample1
```

```
#'bounce-worksheet/lazy-samples1

#'bounce-worksheet/samples1

sample1

(18 ((0.42846830943289227 4.955430158101355) (0.11399011311342733
7.047454084589595) (-1.6760284246021682 3.482088149196809)
(2.5215232534350247 2.5347454541889936) (12.119916165254827
7.446364186029621) (9.842132376151833 0.6555153748336084)
(0.383232923290564 0.8118697300596889) (5.731228040496732
5.245105763879869)))
```

18 balls in the bin.

[Q] Want more balls in the bin. Express this goal using observe.

1) Simulator. 2) Condition. 3) Inference.

[Q] Want more balls in the bin. Express this goal using observe.

1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics2 []
    (let
      [n-bumpers 8
       f (fn [] (list
           (sample (uniform-continuous -5 14))
           (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]

      (observe (normal num-balls 1) 20)
      (list num-balls bs))))
```

[Q] Reduce the number of bumpers.

1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics2 []
    (let
      [n-bumpers • • •
       f (fn [] (list
           (sample (uniform-continuous -5 14))
           (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]

      (observe (normal num-balls 1) 20)
      (list num-balls bs))))
```

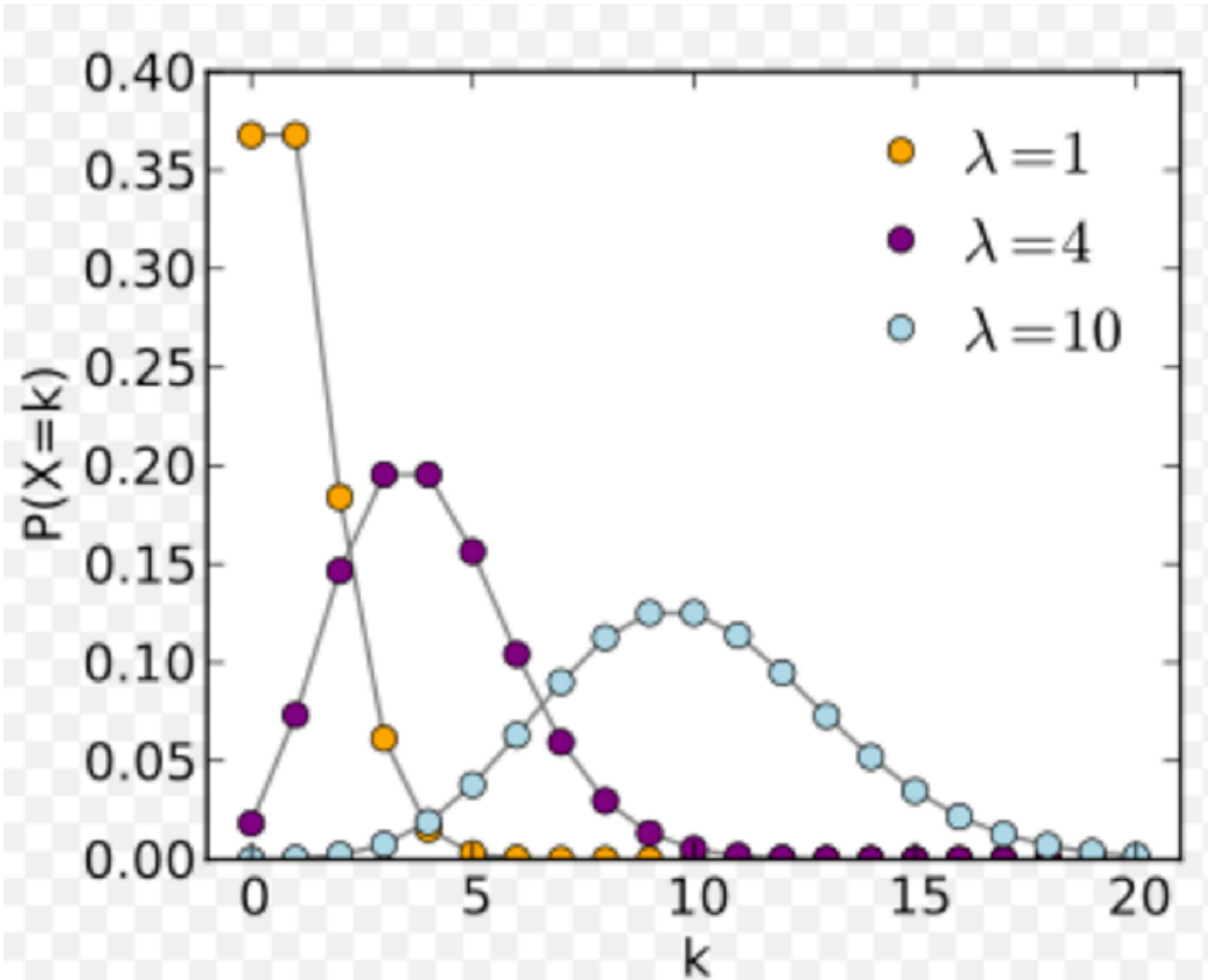
[Q] Reduce the number of bumpers.

Poisson distribution

$$P(k \text{ events in interval}) = e^{-\lambda} \frac{\lambda^k}{k!}$$

- Distribution on non-negative integer k
- Expresses the probability of a given number of events occurring in a fixed time interval, if these events occur with a known constant rate λ, and independently of the time since the last event.
- Mean λ. Peak at λ or λ-1.

Poisson distribution



1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics2 []
    (let
      [n-bumpers (sample (poisson 6))
       f (fn [] (list
               (sample (uniform-continuous -5 14))
               (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]

      (observe (normal num-balls 1) 20)
      (list num-balls bs))))
```

[Q] Reduce the number of bumpers.

1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics2 []
    (let
      [n-bumpers (sample (poisson 6))
       f (fn [] (list
           (sample (uniform-continuous -5 14))
           (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]
      . . . . .
      (observe (normal num-balls 1) 20)
      (list num-balls bs))))
```

[Q] Reduce the number of bumpers.

1) Simulator. 2) Condition. 3) Inference.

```
(with-primitive-procedures
  [create-world simulate-world balls-in-box]
  (defquery physics2 []
    (let
      [n-bumpers (sample (poisson 6))
       f (fn [] (list
            (sample (uniform-continuous -5 14))
            (sample (uniform-continuous 0 10))))
       bs (repeatedly n-bumpers f)
       w0 (create-world bs)
       w1 (simulate-world w0)
       num-balls (balls-in-box w1)]
      (observe (normal n-bumpers 2) 0)
      (observe (normal num-balls 1) 20)
      (list num-balls bs))))
```

[Q] Reduce the number of bumpers.

1) Simulator. 2) Condition. 3) Inference.

```
(def lazy-samples2
  (doquery :lmh physics2 []))
(def samples2
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples2)))))
(defn is-better [x y]
  (let [num-bumpers-less (< (count (second x)) (count (second y)))
        num-balls-more (> (first x) (first y))
        num-balls-equal (= (first x) (first y))
        x-above-threshold (> (first x) 15)
        y-above-threshold (> (first x) 15)]
    (or (and x-above-threshold num-bumpers-less)
        (and num-balls-equal num-bumpers-less)
        num-balls-more)))
(def best-sample2
  (reduce (fn [acc x] (if (is-better x acc) x acc))
          samples2))
best-sample2
```

```
#'bounce-worksheet/lazy-samples2

#'bounce-worksheet/samples2

#'bounce-worksheet/is-better

#'bounce-worksheet/best-sample2

(18 ((3.3372725819978006 8.47981623123972) (6.810994664276837
1.207416460370105 4) (11.516669110855627 2.914632625431299 3)
(-3.905786355145122 6.597352342859228) (0.5808260753916357
0.687127220589358 6)))
```

(W

)

[Q]

1) Simulator. 2) Condition. 3) Inference.

```
(def lazy-samples2
  (doquery :lmh physics2 []))
(def samples2
  (map :result (take-nth 10 (take 2000 (drop 1000 lazy-samples2)))))
(defn is-better [x y]
  (let [num-bumpers-less (< (count (second x)) (count (second y)))
        num-balls-more (> (first x) (first y))
        num-balls-equal (= (first x) (first y))
        x-above-threshold (> (first x) 15)
        y-above-threshold (> (first x) 15)]
    (or (and x-above-threshold num-bumpers-less)
        (and num-balls-equal num-bumpers-less)
        num-balls-more)))
(def best-sample2
  (reduce (fn [acc x] (if (is-better x acc) x acc))
          samples2))
best-sample2
```

```
#'bounce-worksheet/lazy-samples2

#'bounce-worksheet/samples2

#'bounce-worksheet/is-better

#'bounce-worksheet/best-sample2
(18 ((3.3372725819978006 8.47981623123972) (6.810994664276837
1.2074164603701054) (11.516669110855627 2.9146326254312993)
(-3.905786355145122 6.597352342859228) (0.5808260753916357
0.6871272205893586)))
```

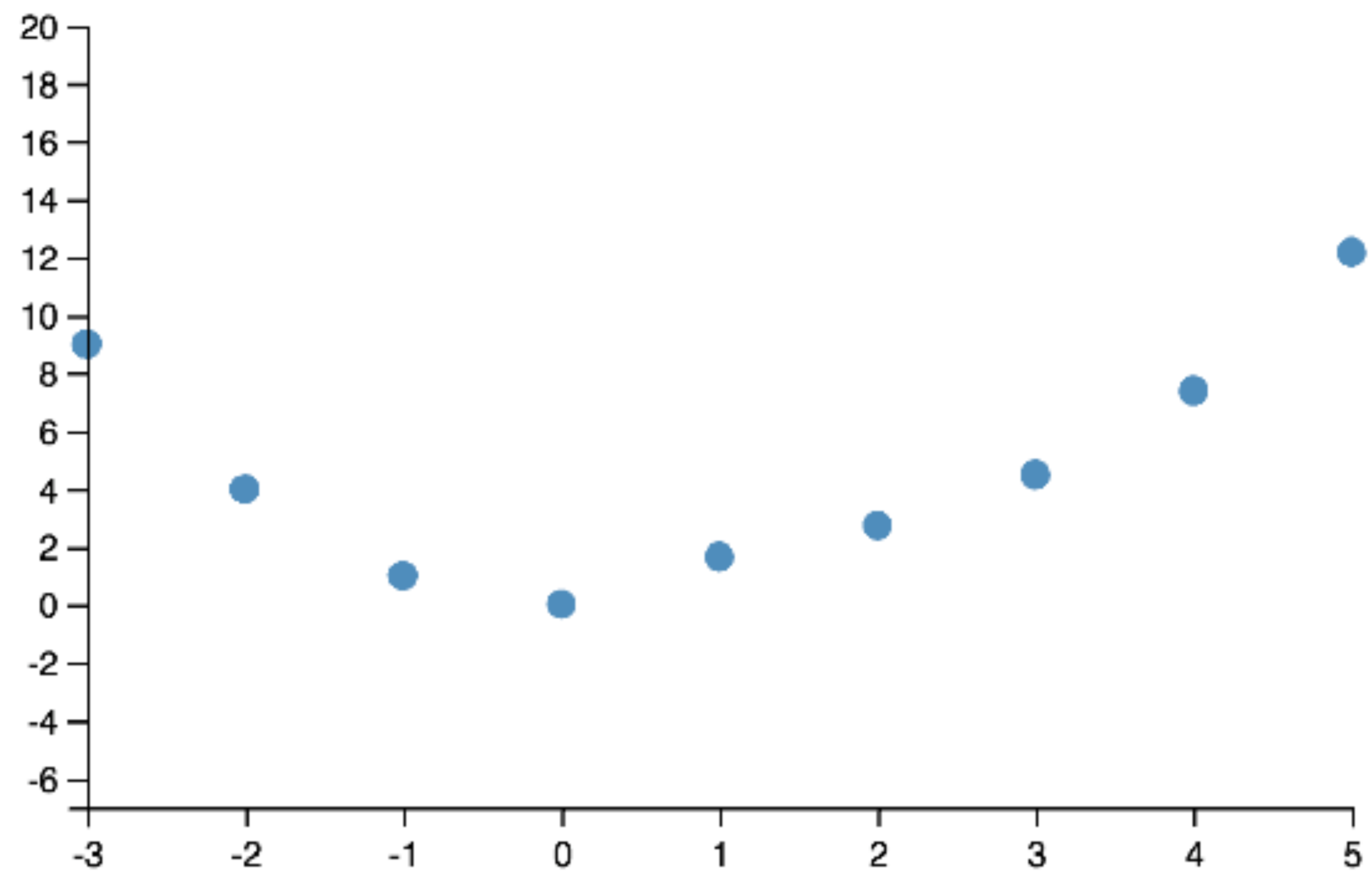
Five bumpers used.

1) Simulator. 2) Condition. 3) Inference.

Baby program induction

Based on Paige, van der Meent & Wood's tutorial at PPAML16 summer school

Baby program induction



[Q] Find a Clojure function that interpolates these data points.

Based on Paige, van der Meent & Wood's tutorial at PPAML16 summer school

Solve program induction via generative modelling

1. Create a simulator.

2. Specify success criteria with conditioning.

3. Solve the problem by posterior inference.

Solve program induction via generative modelling

1. Create a simulator.
   - Generate expressions using probabilistic grammar.
2. Specify success criteria with conditioning.
   - High likelihood if the evaluation of a sampled expression matches data well.
3. Solve the problem by posterior inference.

Solve program induction via generative modelling

1. Create a simulator.
   - Generate expressions using probabilistic grammar.
2. Specify success criteria with conditioning.
   - High likelihood if the evaluation of a sampled expression matches data well.
3. Solve the problem by posterior inference.

Grammar for baby Clojure

```
Prog ::= (fn [x] Ex)

  Ex ::=  Num  |  x  |  BEx

 Num ::= -9  |  -8  | … |  9  |  10

 BEx ::= (+ Ex Ex)  |  (- Ex Ex)  |  (* Ex Ex)
```

Grammar for baby Clojure

Prog ::= (fn [x] Ex)

Ex ::= Num | x | BEx

Num ::= -9 | -8 | … | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

4 Nonterminal symbols.

Grammar for baby Clojure

Prog ::= (fn [x] Ex)

Ex ::= Num | x | BEx

Num ::= -9 | -8 | … | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

4 Nonterminal symbols. Many terminal symbols.

Grammar for baby Clojure

Prog ::= (fn [x] Ex)

Ex ::= Num | x | BEx

Num ::= -9 | -8 | ... | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

4 Nonterminal symbols. Many terminal symbols.

Production rules tell us how to generate programs.

Grammar for baby Clojure

Probabilisitic

Prog ::= (fn [x] Ex)

Ex ::= Num | x | BEx

Num ::= -9 | -8 | ... | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

Grammar for baby Clojure

Probabilisitic

Prog ::= (fn [x] Ex)

Ex ::= Num | x | BEx

Num ::= -9 | -8 | … | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

For each nonterminal, pick a rule probabilistically.

Grammar for baby Clojure

Probabilisitic

1.0

Prog ::= (fn [x] Ex)

Ex ::= Num | x | BEx

Num ::= -9 | -8 | … | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

For each nonterminal, pick a rule probabilistically.

Grammar for baby Clojure

Probabilisitic

1.0

Prog ::= (fn [x] Ex)

0.4 0.4 0.2

Ex ::= Num | x | BEx

Num ::= -9 | -8 | … | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

For each nonterminal, pick a rule probabilistically.

Grammar for baby Clojure

Probabilisitic

1.0

Prog ::= (fn [x] Ex)

0.4 0.4 0.2

Ex ::= Num | x | BEx

0.05 0.05 0.05 0.05

Num ::= -9 | -8 | … | 9 | 10

BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

For each nonterminal, pick a rule probabilistically.

Grammar for baby Clojure

Probabilisitic

1.0
Prog ::= (fn [x] Ex)

0.4 0.4 0.2
Ex ::= Num | x | BEx

0.05 0.05 0.05 0.05
Num ::= -9 | -8 | ... | 9 | 10

0.3 0.3 0.4
BEx ::= (+ Ex Ex) | (- Ex Ex) | (* Ex Ex)

For each nonterminal, pick a rule probabilistically.

Grammar for baby Clojure

Probabilisitic

```
                 1.0
Prog ::= (fn [x] Ex)

          0.4     0.4    0.2
Ex ::=   Num  |   x  |  BEx

          0.05  0.05          0.05  0.05
Num ::=  -9  |  -8  | ... |  9  |  10

              0.3              0.3              0.4
BEx ::= (+ Ex Ex)  |  (- Ex Ex)  |  (* Ex Ex)
```

For each nonterminal, pick a rule probabilistically.

How to represent this prob. grammar in Clojure?

Quoted expressions

- Data structure for Clojure programs.
- Roughly nested lists (and vectors) of symbols & constants.

```
(def e1 (list ‘* (list ‘+ ‘1 ‘x) ‘3))
(def e2 (list ‘fn [‘x] e1))
```

- Can be converted to a program via eval.

```
((eval e2) 10),   ((eval e1) 10)
```

Quoted expressions

- Data structure for Clojure programs.
- Roughly nested lists (and vectors) of symbols & constants.

```
(def e1 (list ‘* (list ‘+ ‘1 ‘x) ‘3))

(def e2 (list ‘fn [‘x] e1))
```

- Can be converted to a program via eval.

```
((eval e2) 10),   ((eval e1) 10)
```

```
(defm gen-e []

     ●    ●    ●    ●    ●        )
```

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, . . ., 10) — (.05, . . ., .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []

        ●  ●  ●  ●  ●            )
```

Anglican's defn.
Anglican functions defined outside of a query.

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]

     ●     ●     ●     ●     ●     ))
```

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))

      • • • • •                   )))
```

Ex: (Num, x, BEx) — (.4, .4, .2)

Num: (-9, …, 10) — (.05, …, .05)

BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))
      (= t 1) ‘x

       • • • • •                         )))
```

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))
      (= t 1) ‘x
      (= t 2) (list

                     • • • • • ))))
```

[Q] Complete this Anglican program.

Ex: (Num, x, BEx) — (.4, .4, .2)	
Num: (-9, …, 10) — (.05, …, .05)	
BEx: (+, -, *) — (.3, .3, .4)	

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))
      (= t 1) ‘x
      (= t 2) (list (sample (categorical
                              {‘+ 0.3,‘- 0.3,‘* 0.4}))
                    (gen-e) (gen-e)))))
```

[Q] Complete this Anglican program.

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
            (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))
      (= t 1) ‘x
      (= t 2) (list (sample (categorical
                              {‘+ 0.3,‘- 0.3,‘* 0.4}))
                    (gen-e) (gen-e)))))
```

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))
      (= t 1) ‘x
      (= t 2) (list (sample (categorical
                               {‘+ 0.3,‘- 0.3,‘* 0.4}))
                    (gen-e) (gen-e)))))
```

```
(defquery grammar []
  (let [prog      • • • • •      ]
    prog))
```

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

```
(defm gen-e []
  (let [t (sample
             (discrete (list 0.4 0.4 0.2)))]
    (cond
      (= t 0) (sample (uniform-discrete -9 11))
      (= t 1) ‘x
      (= t 2) (list (sample (categorical
                             {‘+ 0.3,‘- 0.3,‘* 0.4}))
                    (gen-e) (gen-e)))))

(defquery grammar []
  (let [prog (list ‘fn [‘x] (gen-e))]
    prog))
```

Ex: (Num, x, BEx) — (.4, .4, .2)
Num: (-9, …, 10) — (.05, …, .05)
BEx: (+, -, *) — (.3, .3, .4)

Solve program induction via generative modelling

1. Create a simulator.
    - Generate expressions using probabilistic grammar.
2. Specify success criteria with conditioning.
    - High likelihood if the evaluation of a sampled expression matches data well.
3. Solve the problem by posterior inference.

```
(defm gen-e [] . . . .)


  (defquery grammar []
    (let [prog (list ‘fn [‘x] (gen-e))]




      prog))
```

```
(defm gen-e [] . . . .)



  (defquery grammar [ints outs]
    (let [prog (list ‘fn [‘x] (gen-e))]




      prog))
```

```
(defm gen-e [] . . . .)

(defn evaluate [f-code v] ((eval f-code) v))

(with-primitive-procedures [evaluate]
  (defquery grammar [ints outs]
    (let [prog (list ‘fn [‘x] (gen-e))]







      prog)))
```

```
(defm gen-e [] . . . .)

(defn evaluate [f-code v] ((eval f-code) v))

(with-primitive-procedures [evaluate]
  (defquery grammar [ints outs]
    (let [prog (list ‘fn [‘x] (gen-e))
          f (fn [in out]
              (observe
                 • • • • •
                 out))]
      (map f ins outs)
      prog)))
```

[Q] Complete this.

```
(defm gen-e [] . . . .)

(defn evaluate [f-code v] ((eval f-code) v))

(with-primitive-procedures [evaluate]
  (defquery grammar [ints outs]
    (let [prog (list ‘fn [‘x] (gen-e))
          f (fn [in out]
              (observe
                (normal (evaluate prog in) 1)
                out))]
      (map f ins outs)
      prog)))
```

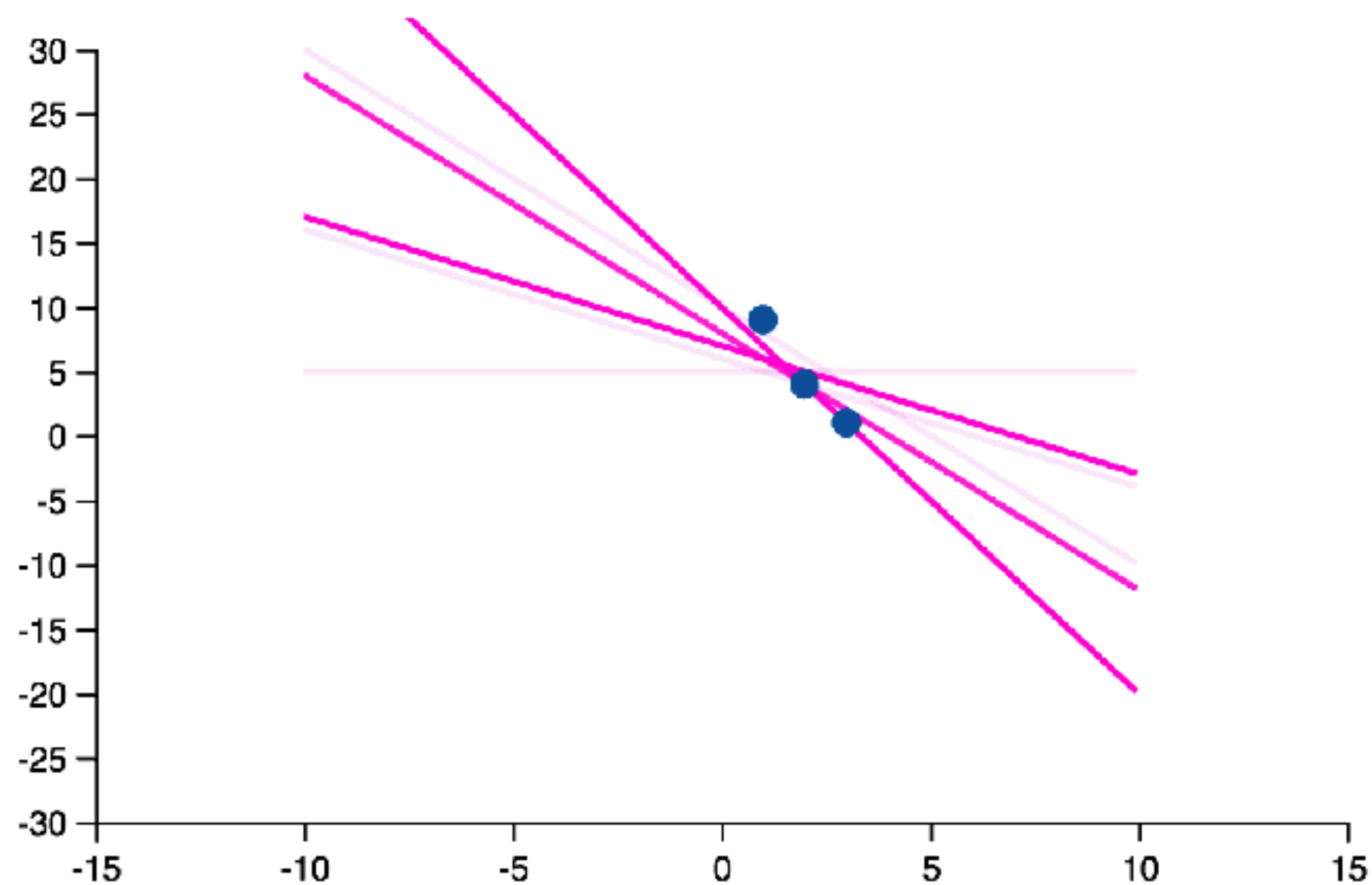
[Q] Complete this.

Solve program induction via generative modelling

1. Create a simulator.
   - Generate expressions using probabilistic grammar.
2. Specify success criteria with conditioning.
   - High likelihood if the evaluation of a sampled expression matches data well.
3. Solve the problem by posterior inference.

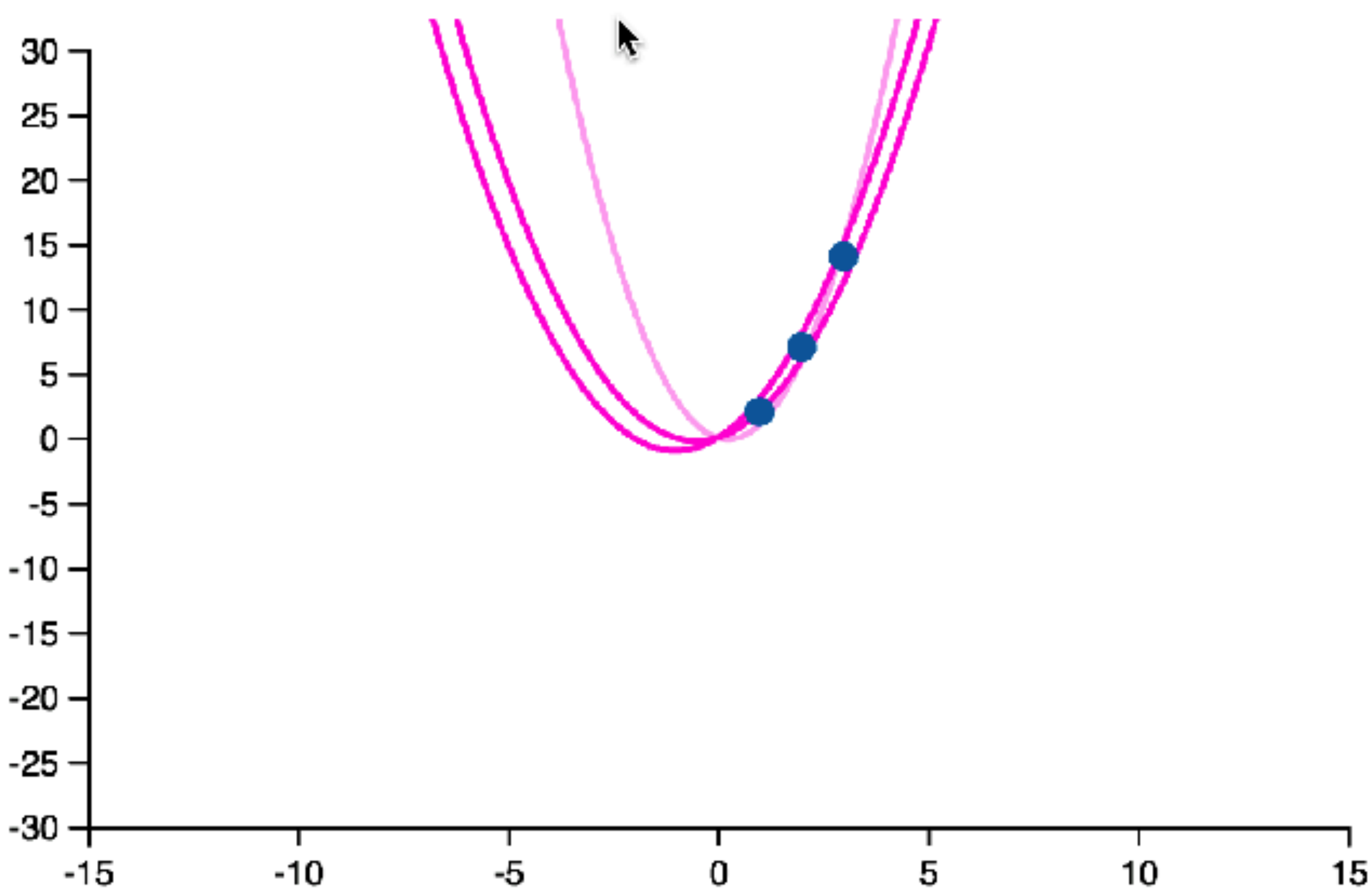
Result with data set 1

```
((fn [x] (+ (* -3 x) 10))
 (fn [x] (+ (* -2 x) 8))
 (fn [x] (+ 1 4))
 (fn [x] (+ (- -1 x) 8))
 (fn [x] (+ (- 0 x) 7)))
```



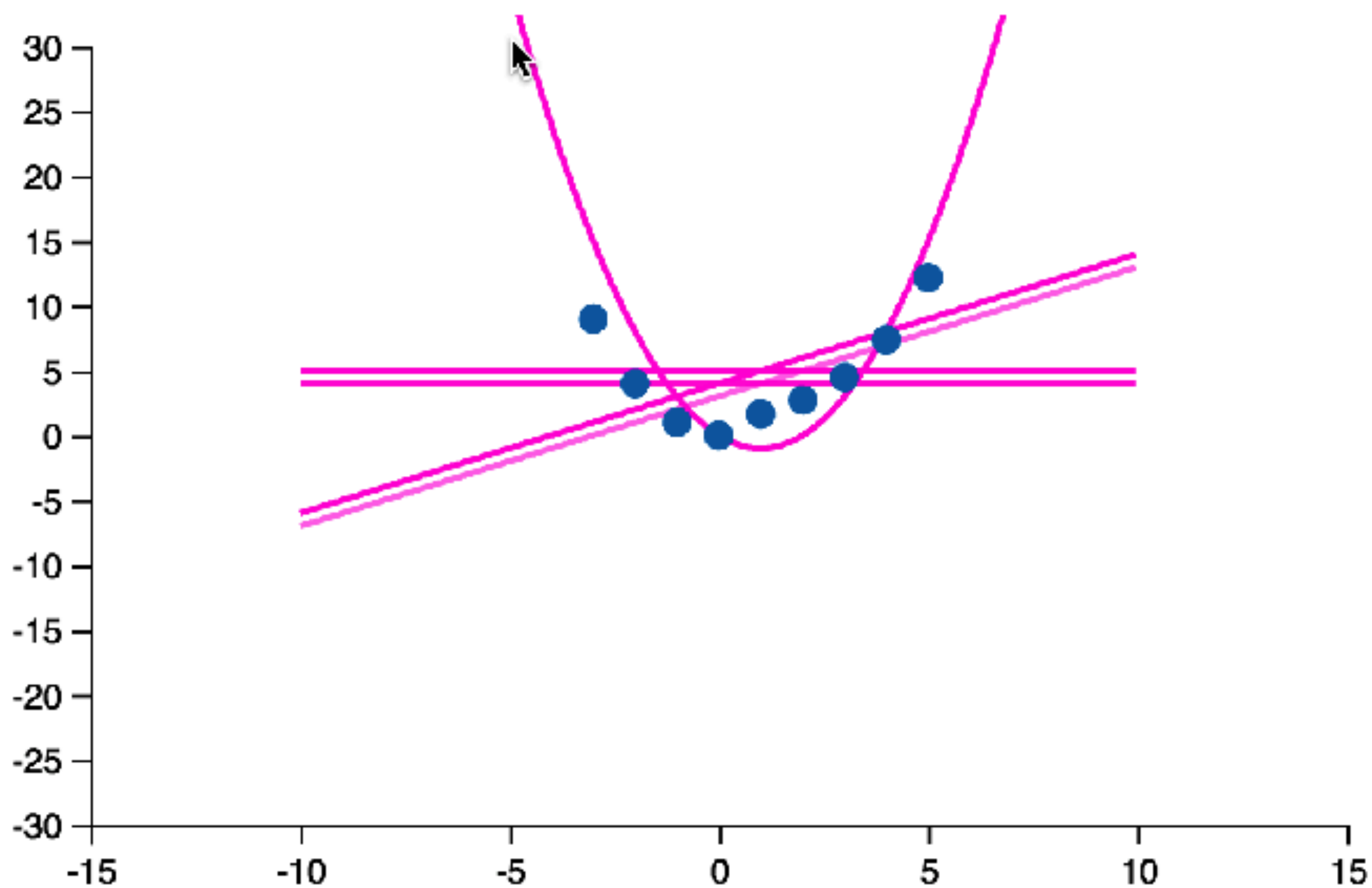
Result with data set 2

```
((fn [x] (* x (+ (* 2 x) -1)))
 (fn [x] (* x (+ x 1)))
 (fn [x] (* x (+ x (+ 2 0))))
 (fn [x] (* (+ (- x 1) (+ x (- x x))) x))
 (fn [x] (* x (+ 2 (* x 1)))))
```



Result with data set 3

```
((fn [x] (* x (- (+ 4 x) (* 6 1))))
 (fn [x] 5)
 (fn [x] (+ 3 x))
 (fn [x] (+ 4 (- x 0)))
 (fn [x] (+ 4 x)))
```



Summary

- The generative approach suggests the specification of a process for generating hypothesis & data.
- This process clarifies hidden assumptions.
- Goes well with probabilistic programming.
- PPLs let us use powerful programming constructs (eval, etc) in modelling.

Information

- I will put gorilla worksheets for 2d physics and program induction in the webpage.
- To run the 2d physics example, you will have to do the following:

1. Copy bounce.clj to anglican-user/src.
2. Copy project.clj to anglican-user.
3. Copy PhysicsLec4.clj to anglican-user/